TURING 图灵程序设计丛书

PACKT PUBLISHING



【荷】Sakis Kasampalis 著　夏永锋 译

精通Python设计模式

Mastering Python Design Patterns

16种基本设计模式，轻松解决软件设计常见问题

借力高效的Python语言，用现实例子展示各模式关键特性

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



Sakis Kasampalis

资深软件工程师，对于多种编程语言和工具都有丰富的经验，原则是在正确的工作上运用正确的工具。Python是他最喜欢的工具之一，因为它十分高效。

夏永锋

毕业于上海交通大学软件学院，曾就职于腾讯上海，现任百度上海研发中心高级研发工程师，爱好编程与翻译。

TURING 图灵程序设计丛书



【荷】Sakis Kasampalis 著 夏永锋 译

精通 Python设计模式

Mastering Python Design Patterns

人民邮电出版社
北 京

图书在版编目（CIP）数据

精通Python设计模式 / （荷）萨基斯·卡萨姆帕里斯（Sakis Kasampalis）著；夏永锋译. -- 北京：人民邮电出版社，2016.7
（图灵程序设计丛书）
ISBN 978-7-115-42803-5

Ⅰ. ①精… Ⅱ. ①萨… ②夏… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2016)第139755号

内 容 提 要

本书分三部分、共 16 章，介绍一些常用的设计模式。第一部分介绍处理对象创建的设计模式，包括工厂模式、建造者模式、原型模式;第二部分介绍处理一个系统中不同实体（类、对象等）之间关系的设计模式，包括外观模式、享元模式等；第三部分介绍处理系统实体之间通信的设计模式，包括责任链模式、观察者模式等。

本书的读者对象为有一定基础的 Python 程序员。

◆ 著　[荷] Sakis Kasampalis
译　夏永锋
责任编辑　朱　巍
执行编辑　杨　琳　赵瑞琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　印刷

◆ 开本：800×1000　1/16
印张：10
字数：236千字　2016年 7 月第 1 版
印数：1 - 3 500册　2016年 7 月北京第 1 次印刷
著作权合同登记号　图字：01-2015-8301号

定价：45.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

版权声明

译　者　序

在我读大学的那几年，设计模式可谓红极一时，各大公司校招面试也几乎都会考设计模式；反观现在，则似乎很少有人聊设计模式的话题。这是因为设计模式过时了吗？还是说它只是一个错误的概念？从个人这几年的开发经验来看，答案是否定的：设计模式并未过时，更不是一个错误的概念。从曾经的“红极一时”到如今的“门可罗雀”，只是说明软件开发行业以更加客观理性的态度来看待设计模式。软件开发领域的技术概念也似乎总是遵循这样的流行度变迁，最终一次又一次地证明不存在“银弹”。

正确看待设计模式的前提是明白什么是设计模式。正如本书一开始就强调的，“设计模式的本质是在已有的方案之上发现更好的方案（而不是全新发明）”。这是一种务实的态度，设计模式并非是某种高大上或者神秘的东西，而是一些常见的软件工程设计问题的最佳实践方案。

那么应该如何学习设计模式？个人认为软件开发技术的学习都应该以实践为前提，只有理解实践过程中遇到的种种问题，才能明白那些技术的本质和目的是什么，因为每种新技术都是因某个/某些问题而出现的；软件开发高手一般都反对新手一开始就一股脑地学习设计模式。有些新手学了点设计模式的理论后，甚至在软件开发过程中生搬硬套，结果适得其反。因此，软件开发人员应该在积累了一定的开发经验之后，再系统地学习设计模式，往往能事半功倍。

现在有些积累了一定开发经验的软件开发人员在谈起设计模式时一脸鄙夷。我想这也不是一种客观务实的态度。软件开发不是简单的累积代码，在实现业务功能的同时应该仔细考虑如何控制软件的复杂度。软件的复杂度分为两个层面：业务逻辑复杂度和代码实现复杂度。对同一个业务系统，不同的软件开发人员会有不同的实现，复杂度也不同；相应地，实现的易理解性、可维护性和可扩展性也不同。软件开发人员应该不断学习如何控制软件的复杂度，学习并恰当地使用设计模式是应对软件复杂度的有效方法。

然而，设计模式并非是固定不变的（如《设计模式：可复用面向对象软件的基础》一书总结的23种模式），使用不同的编程语言来编写代码，需要学习的设计模式也不一样。一方面是因为软件开发领域迅猛发展，一些新的软件工程问题也随之出现；另一方面则是因为新的语言、新的平台会把一些常见设计模式吸收为内置特性。所以，软件开发人员应以实际问题为驱动，不断更新设计模式方面的知识。

本书以Python编程语言为例，针对目前的软件开发领域，分三大类讲解了16种常用的设计模式。使用Python语言编写示例代码，我认为作者主要是考虑到Python的抽象层次高、应用范围广，读者不会被一些实现细节所干扰，从而能快速直接地掌握模式的要领。

全书始终保持务实的态度，列举了大量现实生活的例子和软件开发的例子，并为每个模式提供了完整可运行的示例代码。虽然在书中给出所有示例代码似乎没什么必要，但个人认为作者的用意是希望读者能亲自动手，照着示例代码写一遍并运行，然后看看结果，从而加强学习的效果。

虽然是示例，但作者还是坚持以地道的Python风格编写代码，以此说明不同语言和不同平台要求软件开发人员学习的设计模式也有所不同。另外，开发人员也能从示例代码中学习到一些Python语言的高级特性，所以把本书当作Python开发进阶图书也无不可。

本书是个人正式翻译的第一本书。虽然以前翻译过很多文章，其中有些译文还有一些影响，但毕竟与正式出版有些不同，所以接手本书的翻译工作，我内心是有些忐忑的。为保证翻译的质量，我将翻译过程分为以下几个阶段来进行。

(1) 大致地预读一遍全书，整体上把握原书内容。
(2) 将原书翻译成初稿，此阶段基本保证译文的正确性。
(3) 通读审校初稿，此阶段确保译文的流畅性，以及用词和逻辑的一致性。
(4) 对着译稿，翻译相关图表中的单词；整理示例代码，并确保运行无误。

希望能通过这种方式基本保证译稿的质量。不过因为个人精力有限、能力不足，译稿中可能还会存在疏漏甚至错误之处，敬请谅解。如发现有错误之处，请将问题反馈给出版社，以便在再版时更正。

另外，本书的示例代码已经保存到GitHub的一个代码库（https://github.com/youngsterxyf/mpdp-code）中，如有需要，可以下载。

因个人原因，本书推延了一段时间才得以翻译完成，感谢图灵朱巍老师的体谅。译书是一件费时费力的事情，感谢妻子郑荣的体谅和支持，也感谢公司领导贾磊和同事的支持，谢谢你们！

夏永锋
于上海百度研发中心

前　言

什么是设计模式

软件工程中，设计模式是指软件设计问题的推荐方案。设计模式一般是描述如何组织代码和使用最佳实践来解决常见的设计问题。需谨记在心的一点是：设计模式是高层次的方案，并不关注具体的实现细节，比如算法和数据结构（请参考［GOF95，第13页］和网页［t.cn/RP6HFwi］）。对于正在尝试解决的问题，何种算法和数据结构最优，则是由软件工程师自己把握。

> 如果你不了解［ ］中文字的含义，请暂时先跳到前言的“排版约定”部分，查看一下本书中的引用所遵循的格式。

设计模式最重要的部分可能就是它的名称。给模式起名的好处是大家相互交流时有共同的词汇（请参考［GOF95，第13页］）。因此，如果你提交一些代码进行评审，同行评审者的反馈中提到“我认为这个地方你可以使用一个策略模式来代替……”，即使你不知道或不记得策略模式是什么，也可以立即去查阅。

随着编程语言的演进，一些设计模式（如单例）也随之过时，甚至成了反模式（请参考网页［t.cn/zRoxwyD］），另一些则被内置在编程语言中（如迭代器模式）。另外，也有一些新的模式诞生（比如Borg/Monostate，请参考网页［t.cn/zWOOOZC］和［t.cn/RqrKbBe］）。

关于设计模式的常见误解

关于设计模式有一些误解。第一个误解是，一开始写代码就应该使用设计模式。我们经常能看到开发人员纠结在代码中应该使用哪种设计模式，他们甚至都还没有先尝试一下使用自己的方式解决问题（请参考网页［t.cn/RqrJNDw］和［t.cn/RqrJl0m］）。

这不仅是错误的，而且违背了设计模式的本质。设计模式是在已有的方案之上发现更好的方案（而不是全新发明）。若你一个方案都没有，又何谈找一个更好的呢？先行动起来，用你的技能尽可能漂亮地解决问题。若代码评审者没有反对意见，而你经过一段时间后还是觉得自己的方

案足够漂亮灵活，那也就意味着没必要浪费时间纠结使用哪种模式。你也许还能发现更好的设计模式。谁知道呢，关键是不要为了强迫自己使用已有的设计模式而限制了你的创造力。

第二个误解是设计模式应随处使用。这会导致方案很复杂，夹杂着多余的接口和分层，而其实往往一个更简单直接的方案就足够了。设计模式并不是万能的，仅当代码确实存在坏味道、难以扩展维护时，才有使用的必要。多思考思考你不会需要它（You Aren't Gonna Need It，YAGNI，请参考网页［t.cn/SGw9Ec］）和保持简单直白（Keep It Simple Stupid，KISS，请参考网页［t.cn/RqrKMW4］）原则。随处使用设计模式与过早优化一样，都是误入歧途（请参考网页［t.cn/ShMKfD］）。

设计模式与 Python

本书主要介绍Python实现的设计模式。与畅销设计模式书籍中大多使用的常见编程语言（通常是Java，请参考［FFBS04］；或C++，请参考［GOF95］）不同，Python支持动态类型（duck-typing），函数是一等公民，并且一些模式（例如，迭代器和修饰器）是内置特性。本书旨在演示最基本的设计模式，并非历史记载的所有模式（请参考网页［t.cn/RqrKbBe］）。代码示例也使用合适的Python惯用写法（请参考网页［t.cn/hTfLt］）。如果你还不熟悉Python之禅，那现在就打开Python交互模式，执行import this。Python之禅趣味十足又意义深远。

本书内容

第一部分，创建型模式，介绍处理对象创建的设计模式。

- 第1章，工厂模式　介绍如何使用工厂设计模式（工厂方法和抽象工厂）来初始化对象，并说明与直接实例化对象相比，使用工厂设计模式的优势。
- 第2章，建造者模式　对于由多个相关对象构成的对象，介绍如何简化其创建过程。
- 第3章，原型模式　介绍如何通过完全复制（也就是克隆）一个已有对象来创建一个新对象。

第二部分，结构型模式，介绍处理一个系统中不同实体（类、对象等）之间关系的设计模式。

- 第4章，适配器模式　介绍如何以最小的改变实现已有代码与外来接口（例如，一个外部代码库）的兼容。
- 第5章，修饰器模式　介绍如何无需使用继承也能增强对象的功能。
- 第6章，外观模式　介绍如何创建单个入口点来隐藏系统的复杂性。
- 第7章，享元模式　介绍如何通过复用一个对象池中的对象来提高内存利用率及应用性能。
- 第8章，模型-视图-控制器模式　介绍如何避免业务逻辑与用户界面代码的耦合，提高应用的可维护性。

- 第9章，代理模式　介绍如何增加额外的保护层，提高应用的安全性。

第三部分，行为型模式，介绍处理系统实体之间通信的设计模式。

- 第10章，责任链模式　介绍如何向多个接收者发送请求。
- 第11章，命令模式　介绍如何让应用能够取消已经执行的操作。
- 第12章，解释器模式　介绍如何基于Python创建一种简单的语言，便于领域专家使用，而无需学习Python编程。
- 第13章，观察者模式　介绍如何在对象发生变化时，通知已注册的相关者。
- 第14章，状态模式　介绍如何创建一个状态机以对问题进行建模，并说明这种技术的优势。
- 第15章，策略模式　介绍如何基于某些输入标准（例如，元素大小）在程序运行期间从多个可用算法中选择一个。
- 第16章，模板模式　介绍如何明确区分一个算法的通用与不通用部分，以避免不必要的代码复制。

阅读准备

书中的代码仅用Python 3编写。Python 3在很多方面与Python 2.x不兼容（请参考网页［t.cn/Rw8Ycjs］）。虽然代码是使用Python 3.4.0进行测试的，但Python 3.3.0应该也可以，因为Python 3.3.0和Python 3.4.0之间并没有语法上的差别（请参考网页［t.cn/RqrK1eX］）。一般来说，如果你从www.python.org下载安装最新的Python 3版本，那么运行示例代码应该不会有问题。示例代码中使用的多数模块/库是Python 3自带的。如果有示例要求安装额外的模块，在相关代码之前会给出如何安装的说明。

读者对象

本书适合具备一定经验同时又对以地道的Python代码实现设计模式感兴趣的Python开发人员。使用其他语言的开发人员，如果对Python感兴趣，也能从中获益不少，但最好先阅读一些材料，了解一下Python的基本知识（请参考网页［t.cn/hTfLt］和网页［t.cn/hp20G］）。

排版约定

在书中，你会发现许多文本样式，用于区分不同种类的信息。以下举例说明，并解释其含义。

代码、用户输入、推特用户定位会使用等宽的代码字体，如下面的句子中所示："我们将使用Python发行版自带的两个库（`xml.etree.ElementTree`和`json`）来处理XML和JSON。"

代码块的版式如下所示。

```
@property
def parsed_data(self):
    return self.data
```

当希望你关注代码块中某个特别部分时，相关的行或项目会以粗体显示，如下所示。

```
@property
def parsed_data(self):
    return self.data
```

任何命令行输入输出都按如下方式编写。

```
>>> python3 factory_method.py
```

新术语和重要的单词以楷体显示。在菜单或对话框等屏幕上看到的单词会保留原英文，如“点击Next按钮转到下一屏”。

> 警告或重要的注意事项会这样显示。

> 提示和小窍门会这样显示。

书籍引用遵循格式［作者，页码］。例如，［GOF95，第10页］是指引用GOF（《设计模式：可复用面向对象软件的基础》）一书的第10页。

Web引用遵循格式［t.cn/shortened］。可以将这些缩短的URL地址键入或复制到Web浏览器中，按Enter键后会跳转到实际的Web引用（通常也更长，也许会更丑）。例如，在Web浏览器地址栏中键入t.cn/hTfLt，按Enter键后会跳转到http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html。

读者反馈

我们始终欢迎来自读者的反馈。关于本书，如果你有一些想法，无论是喜欢还是不喜欢，都可以告诉我们。我们非常重视读者反馈，因为这能帮助我们开发一些让读者充分受益的出版物。

一般的反馈，只要发送电子邮件到feedback@packtpub.com并在消息标题中提及书名即可。

如果你擅长某个主题，并有兴趣写本书或者为某本书作出贡献，请阅读作者指南www.packtpub.com/authors。

客户支持

既然你购买了一本Packt出版的书籍，我们会提供很多服务，让你获得最大的购买利益。

下载示例代码

凡是通过http://www.packtpub.com网站账户购买的Packt书籍，都可以在网上下载相应的示例代码文件。如果你是在其他地方购买本书的，则可以访问http://packtpub.com/support，注册之后，相关文件会直接通过电子邮件发送给你。

勘误

虽然我们会全力确保书籍内容的准确性，但错误仍然在所避免。如果你在某本书中发现错误，无论是文本错误还是代码错误，都请报告给我们，对此我们将万分感激。这样不仅能消除其他读者的疑虑，也能帮助我们提升本书后续版本的质量。如果你发现任何错误，请访问http://www.packtpub.com/submit-errata进行报告，选择相应图书，单击Errata Submission Form链接，并输入勘误的详细信息。一旦勘误得到证实，我们就会接受你的提交，并将勘误上传到站点或添加到对应书名的勘误一节下面的已有勘误表中。

要查看之前提交的勘误，可以访问https://www.packtpub.com/books/content/support并在搜索框内输入书名。需要的信息会显示在Errata部分的下面。

反盗版

对所有媒体来说，互联网盗版行为都是一直存在的问题。在Packt，我们严格保护版权和许可证。如果你在互联网上发现任何形式的我们出版物的非法复制品，请立即把网址或站点名称提供给我们，以便我们进行补救。

请通过copyright@packtpub.com联系我们，提供可疑盗版资料的链接。

感谢你帮助保护我们的作者，让我们能够为你提供有价值的内容。

疑问解答

对于本书的任何方面，如果你有问题，可以通过question@packtpub.com联系我们，我们将尽力解决。

电子书

扫描如下二维码，即可购买本书电子版。

目　　录

第一部分　创建型模式

第二部分　结构型模式

第三部分 行为型模式

Part 1

第一部分

创建型模式

本部分内容

第 1 章

工厂模式

创建型设计模式处理对象创建相关的问题（请参考网页［t.cn/RqBoSiu］），目标是当直接创建对象（在Python中是通过`__init__()`函数实现的，请参考网页［t.cn/RqB3mDM］和［Lott14，第26页］）不太方便时，提供更好的方式。

在工厂设计模式中，客户端[1]可以请求一个对象，而无需知道这个对象来自哪里；也就是，使用哪个类来生成这个对象。工厂背后的思想是简化对象的创建。与客户端自己基于类实例化直接创建对象相比，基于一个中心化函数来实现，更易于追踪创建了哪些对象（请参考［Eckel08，第187页］）。通过将创建对象的代码和使用对象的代码解耦，工厂能够降低应用维护的复杂度（请参考［Zlobin13，第30页］）。

工厂通常有两种形式：一种是*工厂方法*（Factory Method），它是一个方法（或以地道的Python术语来说，是一个函数），对不同的输入参数返回不同的对象（请参考网页［t.cn/RqB1yx2］）；第二种是抽象工厂，它是一组用于创建一系列相关事物对象的工厂方法（请参考［GOF95，第100页］和网页［t.cn/RqB1tZS］）。

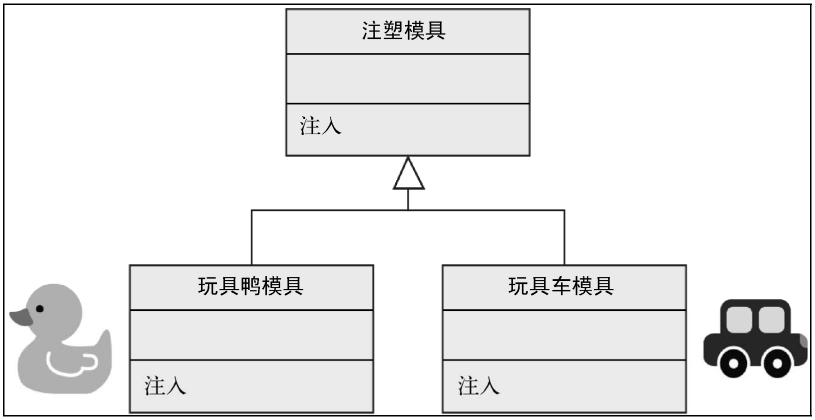
1.1 工厂方法

在工厂方法模式中，我们执行单个函数，传入一个参数（提供信息表明我们想要什么），但并不要求知道任何关于对象如何实现以及对象来自哪里的细节。

1.1.1 现实生活的例子

现实中用到工厂方法模式思想的一个例子是塑料玩具制造。制造塑料玩具的压塑粉都是一样的，但使用不同的塑料模具就能产出不同的外形。比如，有一个工厂方法，输入是目标外形（鸭子或小车）的名称，输出则是要求的塑料外形。下图展示的是玩具制造案例，该案例源自网站www.sourcemaking.com，请参考网页［t.cn/RqB1yx2］。

① 本书中涉及的“客户端”一词是指调用方，并非网络CS结构中的C。——译者注



1.1.2 软件的例子

Django框架使用工厂方法模式来创建表单字段。Django的`forms`模块支持不同种类字段(`CharField`、`EmailField`)的创建和定制(`max_length`、`required`),请参考网页[t.cn/Rqr9qD7]。

1.1.3 应用案例

如果因为应用创建对象的代码分布在多个不同的地方,而不是仅在一个函数/方法中,你发现没法跟踪这些对象,那么应该考虑使用工厂方法模式(请参考[Eckel08,第187页])。工厂方法集中地在一个地方创建对象,使对象跟踪变得更容易。注意,创建多个工厂方法也完全没有问题,实践中通常也这么做,对相似的对象创建进行逻辑分组,每个工厂方法负责一个分组。例如,有一个工厂方法负责连接到不同的数据库(MySQL、SQLite),另一个工厂方法负责创建要求的几何对象(圆形、三角形),等等。

若需要将对象的创建和使用解耦,工厂方法也能派上用场。创建对象时,我们并没有与某个特定类耦合/绑定到一起,而只是通过调用某个函数来提供关于我们想要什么的部分信息。这意味着修改这个函数比较容易,不需要同时修改使用这个函数的代码(请参考[Zlobin13,第30页])。

另外一个值得一提的应用案例与应用性能及内存使用相关。工厂方法可以在必要时创建新的对象,从而提高性能和内存使用率(请参考[Zlobin13,第28页])。若直接实例化类来创建对象,那么每次创建新对象就需要分配额外的内存(除非这个类内部使用了缓存,一般情况下不会这样)。用行动说话,下面的代码(文件id.py)对同一个类`A`创建了两个实例,并使用函数`id()`比较它们的内存地址。输出中也会包含地址,便于检查地址是否正确。内存地址不同就意味着创建了两个不同的对象。

```
class A(object):
    pass

if __name__ == '__main__':
    a = A()
    b = A()

    print(id(a) == id(b))
    print(a, b)
```

在我的计算机上执行id.py，输出的内容如下所示。

```
>>> python3 id.py
False

<__main__.A object at 0x7f5771de8f60> <__main__.A object at
0x7f5771df2208>
```

注意，你执行这个代码文件看到的地址会与我看到的不一样，因为这依赖程序运行时内存的布局和分配。但结果中有一点肯定是一样的，那就是两个地址不同。在Python Read-Eval-Print Loop（REPL）模式（即交互式提示模式）下编写运行这段代码时会出现例外，但这只是交互模式特有的优化，并不常见。

1.1.4　实现

数据来源可以有多种形式。存取数据的文件主要有两种分类：人类可读文件和二进制文件。人类可读文件的例子有：XML、Atom、YAML和JSON。二进制文件的例子则有SQLite使用的.sq3文件格式，及用于听音乐的.mp3文件格式。

以下例子将关注两种流行的人类可读文件格式：XML和JSON。虽然人类可读文件解析起来通常比二进制文件更慢，但更易于数据交换、审查和修改。基于这种考虑，建议优先使用人类可读文件，除非有其他限制因素不允许使用这类格式（主要的限制包括性能不可接受以及专有的二进制格式）。

在当前这个问题中，我们有一些输入数据存储在一个XML文件和一个JSON文件中，要对这两个文件进行解析，获取一些信息。同时，希望能够对这些（以及将来涉及的所有）外部服务进行集中式的客户端连接。我们使用工厂方法来解决这个问题。虽然仅以XML和JSON为例，但为更多的服务添加支持也很简单。

首先，来看一看数据文件。基于Wikipedia例子（请参考网页[t.cn/RqB1Y9F]）的XML文件person.xml包含个人信息（`firstName`、`lastName`、`gender`等），如下所示。

```
<persons>
  <person>
    <firstName>John</firstName>
```

```
      <lastName>Smith</lastName>
      <age>25</age>
      <address>
        <streetAddress>21 2nd Street</streetAddress>
        <city>New York</city>
        <state>NY</state>
        <postalCode>10021</postalCode>
      </address>
      <phoneNumbers>
        <phoneNumber type="home">212 555-1234</phoneNumber>
        <phoneNumber type="fax">646 555-4567</phoneNumber>
      </phoneNumbers>
      <gender>
        <type>male</type>
      </gender>
    </person>
    <person>
      <firstName>Jimy</firstName>
      <lastName>Liar</lastName>
      <age>19</age>
      <address>
        <streetAddress>18 2nd Street</streetAddress>
        <city>New York</city>
        <state>NY</state>
        <postalCode>10021</postalCode>
      </address>
      <phoneNumbers>
        <phoneNumber type="home">212 555-1234</phoneNumber>
      </phoneNumbers>
      <gender>
        <type>male</type>
      </gender>
    </person>
    <person>
      <firstName>Patty</firstName>
      <lastName>Liar</lastName>
      <age>20</age>
      <address>
        <streetAddress>18 2nd Street</streetAddress>
        <city>New York</city>
        <state>NY</state>
        <postalCode>10021</postalCode>
      </address>
      <phoneNumbers>
        <phoneNumber type="home">212 555-1234</phoneNumber>
        <phoneNumber type="mobile">001 452-8819</phoneNumber>
      </phoneNumbers>
      <gender>
        <type>female</type>
      </gender>
    </person>
  </persons>
```

JSON文件donut.json来自Adobe的GitHub账号（请参考网页［t.cn/RqB1udG］），包含甜甜圈（donut）信息（type、单位价格ppu、topping等），如下所示。

```
[
  {
    "id": "0001",
    "type": "donut",
    "name": "Cake",
    "ppu": 0.55,
    "batters": {
      "batter": [
        { "id": "1001", "type": "Regular" },
        { "id": "1002", "type": "Chocolate" },
        { "id": "1003", "type": "Blueberry" },
        { "id": "1004", "type": "Devil's Food" }
      ]
    },
    "topping": [
      { "id": "5001", "type": "None" },
      { "id": "5002" "type": "Glazed" },
      { "id": "5005", "type": "Sugar" },
      { "id": "5007", "type": "Powdered Sugar" },
      { "id": "5006", "type": "Chocolate with Sprinkles" },
      { "id": "5003", "type": "Chocolate" },
      { "id": "5004", "type": "Maple" }
    ]
  },
  {
    "id": "0002",
    "type": "donut",
    "name": "Raised",
    "ppu": 0.55,
    "batters": {
      "batter": [
        { "id": "1001", "type": "Regular" }
      ]
    },
    "topping": [
      { "id": "5001", "type": "None" },
      { "id": "5002", "type": "Glazed" },
      { "id": "5005", "type": "Sugar" },
      { "id": "5003", "type": "Chocolate" },
      { "id": "5004", "type": "Maple" }
    ]
  },
  {
    "id": "0003",
    "type": "donut",
    "name": "Old Fashioned",
    "ppu": 0.55,
    "batters": {
      "batter": [
        { "id": "1001", "type": "Regular" },
```

```
            { "id": "1002", "type": "Chocolate" }
          ]
        },
        "topping": [
          { "id": "5001", "type": "None" },
          { "id": "5002", "type": "Glazed" },
          { "id": "5003", "type": "Chocolate" },
          { "id": "5004", "type": "Maple" }
        ]
      }
    ]
```

我们将使用Python发行版自带的两个库（xml.etree.ElementTree和json）来处理XML和JSON，如下所示。

```
import xml.etree.ElementTree as etree
import json
```

类JSONConnector解析JSON文件，通过parsed_data()方法以一个字典（dict）的形式返回数据。修饰器property使parsed_data()显得更像一个常规的变量，而不是一个方法，如下所示。

```
class JSONConnector:

    def __init__(self, filepath):
        self.data = dict()
        with open(filepath, mode='r', encoding='utf-8') as f:
            self.data = json.load(f)

    @property
    def parsed_data(self):
        return self.data
```

类XMLConnector解析 XML 文件，通过parsed_data()方法以xml.etree.Element列表的形式返回所有数据，如下所示。

```
class XMLConnector:

    def __init__(self, filepath):
        self.tree = etree.parse(filepath)

    @property
    def parsed_data(self):
        return self.tree
```

函数connection_factory是一个工厂方法，基于输入文件路径的扩展名返回一个JSONConnector或XMLConnector的实例，如下所示。

```
def connector_factory(filepath):
    if filepath.endswith('json'):
        connector = JSONConnector
```

```
    elif filepath.endswith('xml'):
        connector = XMLConnector
    else:
        raise ValueError('Cannot connect to {}'.format(filepath))
    return connector(filepath)
```

函数`connect_to()`对`connection_factory()`进行包装，添加了异常处理，如下所示。

```
def connect_to(filepath):
    factory = None
    try:
        factory = connection_factory(filepath)
    except ValueError as ve:
        print(ve)
    return factory
```

函数`main()`演示如何使用工厂方法设计模式。第一部分是确认异常处理是否有效，如下所示。

```
def main():
    sqlite_factory = connect_to('data/person.sq3')
```

接下来的部分演示如何使用工厂方法处理XML文件。XPath用于查找所有包含姓（last name）为`Liar`的`person`元素。对于每个匹配到的元素，展示其基本的姓名和电话号码信息，如下所示。

```
    xml_factory = connect_to('data/person.xml')
    xml_data = xml_factory.parsed_data()
    liars = xml_data.findall(".//{person}[{lastName}='{}']".format('Liar'))
    print('found: {} persons'.format(len(liars)))
    for liar in liars:
        print('first name: {}'.format(liar.find('firstName').text))
        print('last name: {}'.format(liar.find('lastName').text))
        [print('phone number ({}):'.format(p.attrib['type']),
        p.text) for p in liar.find('phoneNumbers')]
```

最后一部分演示如何使用工厂方法处理JSON文件。这里没有模式匹配，因此所有甜甜圈的`name`、`price`和`topping`如下所示。

```
    json_factory = connect_to('data/donut.json')
    json_data = json_factory.parsed_data
    print('found: {} donuts'.format(len(json_data)))
    for donut in json_data:
        print('name: {}'.format(donut['name']))
        print('price: ${}'.format(donut['ppu']))
        [print('topping: {} {}'.format(t['id'], t['type'])) for t
        in donut['topping']]
```

为便于整体理解，下面给出工厂方法实现（factory_method.py）的完整代码。

```
import xml.etree.ElementTree as etree
import json
```

```
class JSONConnector:
    def __init__(self, filepath):
        self.data = dict()
        with open(filepath, mode='r', encoding='utf-8') as f:
            self.data = json.load(f)

    @property
    def parsed_data(self):
        return self.data

class XMLConnector:
    def __init__(self, filepath):
        self.tree = etree.parse(filepath)

    @property
    def parsed_data(self):
        return self.tree

def connection_factory(filepath):
    if filepath.endswith('json'):
        connector = JSONConnector
    elif filepath.endswith('xml'):
        connector = XMLConnector
    else:
        raise ValueError('Cannot connect to {}'.format(filepath))
    return connector(filepath)

def connect_to(filepath):
    factory = None
    try:
        factory = connection_factory(filepath)
    except ValueError as ve:
        print(ve)
    return factory

def main():
    sqlite_factory = connect_to('data/person.sq3')
    print()

    xml_factory = connect_to('data/person.xml')
    xml_data = xml_factory.parsed_data
    liars = xml_data.findall(".//{}[{}='{}']".format('person',
    'lastName', 'Liar'))
    print('found: {} persons'.format(len(liars)))
    for liar in liars:
        print('first name: {}'.format(liar.find('firstName').text))
        print('last name: {}'.format(liar.find('lastName').text))
        [print('phone number ({})'.format(p.attrib['type']),
        p.text) for p in liar.find('phoneNumbers')]
    print()

    json_factory = connect_to('data/donut.json')
    json_data = json_factory.parsed_data
    print('found: {} donuts'.format(len(json_data)))
```

```
    for donut in json_data:
    print('name: {}'.format(donut['name']))
    print('price: ${}'.format(donut['ppu']))
    [print('topping: {} {}'.format(t['id'], t['type'])) for t in donut['topping']]

if __name__ == '__main__':
    main()
```

该程序的输出如下所示。

```
>>> python3 factory_method.py
Cannot connect to data/person.sq3

found: 2 persons
first name: Jimy
last name: Liar
phone number (home): 212 555-1234
first name: Patty
last name: Liar
phone number (home): 212 555-1234
phone number (mobile): 001 452-8819

found: 3 donuts
name: Cake
price: $0.55
topping: 5001 None
topping: 5002 Glazed
topping: 5005 Sugar
topping: 5007 Powdered Sugar
topping: 5006 Chocolate with Sprinkles
topping: 5003 Chocolate
topping: 5004 Maple
name: Raised
price: $0.55
topping: 5001 None
topping: 5002 Glazed
topping: 5005 Sugar
topping: 5003 Chocolate
topping: 5004 Maple
name: Old Fashioned
price: $0.55
topping: 5001 None
topping: 5002 Glazed
topping: 5003 Chocolate
topping: 5004 Maple
```

注意，虽然`JSONConnector`和`XMLConnector`拥有相同的接口，但是对于`parsed_data()`返回的数据并不是以统一的方式进行处理。对于每个连接器，需使用不同的Python代码来处理。若能对所有连接器应用相同的代码当然最好，但是在多数时候这是不现实的，除非对数据使用某种共同的映射，这种映射通常是由外部数据提供者提供。即使假设可以使用相同的代码来处理XML和JSON文件，当需要支持第三种格式（例如，SQLite）时，又该对代码作哪些改变呢？找

一个SQlite文件或者自己创建一个，尝试一下。

像现在这样，代码并未禁止直接实例化一个连接器。如果要禁止直接实例化，是否可以实现？试试看。

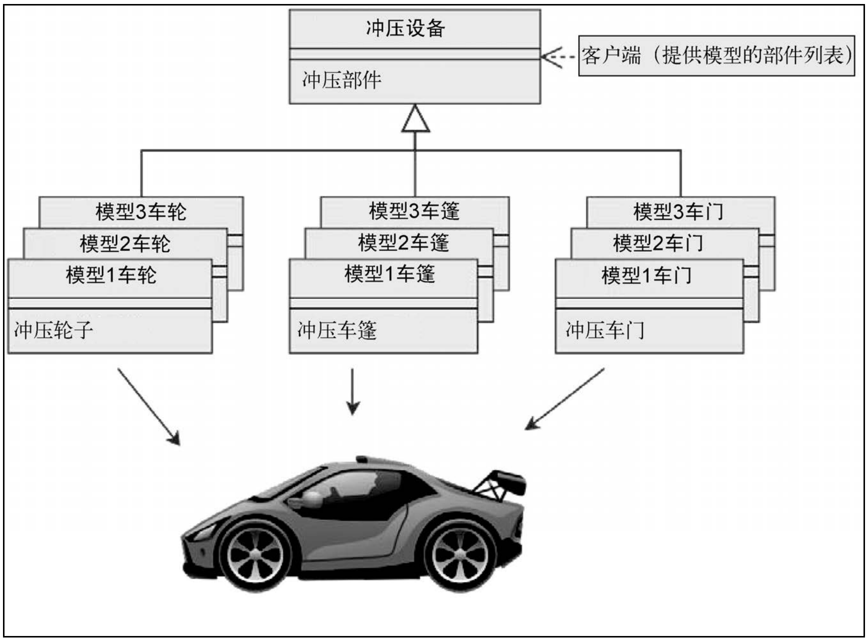
Python中的函数可以内嵌类。

1.2　抽象工厂

抽象工厂设计模式是抽象方法的一种泛化。概括来说，一个抽象工厂是（逻辑上的）一组工厂方法，其中的每个工厂方法负责产生不同种类的对象（请参考［Eckel08，第193页］）。

1.2.1　现实生活的例子

汽车制造业应用了抽象工厂的思想。冲压不同汽车模型的部件（车门、仪表盘、车篷、挡泥板及反光镜等）所使用的机件是相同的。机件装配起来的模型随时可配置，且易于改变。从下图我们能看到汽车制造业抽象工厂的一个例子，该图由www.sourcemaking.com提供（请参考网页［t.cn/RqB1tZS］）。



1.2.2　软件的例子

程序包django_factory是一个用于在测试中创建Django模型的抽象工厂实现，可用来为支持测试专有属性的模型创建实例。这能让测试代码的可读性更高，且能避免共享不必要的代码，故有其存在的价值（请参考网页［t.cn/RqBBvcw］）。

1.2.3　应用案例

因为抽象工厂模式是工厂方法模式的一种泛化，所以它能提供相同的好处：让对象的创建更容易追踪；将对象创建与使用解耦；提供优化内存占用和应用性能的潜力。

这样会产生一个问题：我们怎么知道何时该使用工厂方法，何时又该使用抽象工厂？答案是，通常一开始时使用工厂方法，因为它更简单。如果后来发现应用需要许多工厂方法，那么将创建一系列对象的过程合并在一起更合理，从而最终引入抽象工厂。

抽象工厂有一个优点，在使用工厂方法时从用户视角通常是看不到的，那就是抽象工厂能够通过改变激活的工厂方法动态地（运行时）改变应用行为。一个经典例子是能够让用户在使用应用时改变应用的观感（比如，Apple风格和Windows风格等），而不需要终止应用然后重新启动（请参考［GOF95，第99页］）。

1.2.4　实现

为演示抽象工厂模式，我将重新使用*Python 3 Patterns & Idioms*（Bruce Eckel著）一书中的一个例子（请参考［Eckel08，第193页］），它是我个人最喜欢的例子之一。想象一下，我们正在创造一个游戏，或者想在应用中包含一个迷你游戏让用户娱乐娱乐。我们希望至少包含两个游戏，一个面向孩子，一个面向成人。在运行时，基于用户输入，决定该创建哪个游戏并运行。游戏的创建部分由一个抽象工厂维护。

从孩子的游戏说起，我们将该游戏命名为FrogWorld。主人公是一只青蛙，喜欢吃虫子。每个英雄都需要一个好名字，在我们的例子中，这个名字在运行时由用户给定。方法`interact_with()`用于描述青蛙与障碍物（比如，虫子、迷宫或其他青蛙）之间的交互，如下所示。

```
class Frog:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
```

```
        print('{} the Frog encounters {} and {}!'.format(self,
        obstacle, obstacle.action()))
```

障碍物可以有多种，但对于我们的例子，可以仅仅是虫子。当青蛙遇到一只虫子，只支持一种动作，那就是吃掉它！

```
class Bug:
    def __str__(self):
        return 'a bug'

    def action(self):
        return 'eats it'
```

类FrogWorld是一个抽象工厂，其主要职责是创建游戏的主人公和障碍物。区分创建方法并使其名字通用（比如，make_character()和make_obstacle()），这让我们可以动态改变当前激活的工厂（也因此改变了当前激活的游戏），而无需进行任何代码变更。在一门静态语言中，抽象工厂是一个抽象类/接口，具备一些空方法，但在Python中无需如此，因为类型是在运行时检测的（请参考［Eckel08，第195页］和网页［t.cn/h47Rs9］），如下所示。

```
class FrogWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t------ Frog World-------'

    def make_character(self):
        return Frog(self.player_name)

    def make_obstacle(self):
        return Bug()
```

WizardWorld游戏也类似。在故事中唯一的区别是男巫战怪兽（如兽人）而不是吃虫子！

```
class Wizard:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
        print('{} the Wizard battles against {} and {}!'.format(self,
        obstacle, obstacle.action()))

class Ork:
    def __str__(self):
        return 'an evil ork'

    def action(self):
```

```
        return 'kills it'

class WizardWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t------ Wizard World -------'

    def make_character(self):
        return Wizard(self.player_name)

    def make_obstacle(self):
        return Ork()
```

类GameEnvironment是我们游戏的主入口。它接受factory作为输入，用其创建游戏的世界。方法play()则会启动hero和obstacle之间的交互，如下所示。

```
class GameEnvironment:
    def __init__(self, factory):
        self.hero = factory.make_character()
        self.obstacle = factory.make_obstacle()

    def play(self):
        self.hero.interact_with(self.obstacle)
```

函数validate_age()提示用户提供一个有效的年龄。如果年龄无效，则会返回一个元组，其第一个元素设置为False。如果年龄没问题，元素的第一个元素则设置为True，但我们真正关心的是元素的第二个元素，也就是用户提供的年龄，如下所示。

```
def validate_age(name):
    try:
        age = input('Welcom {}. How old are you? '.format(name))
        age = int(age)
    except ValueError as err:
        print("Age {} is invalid, please try again...".format(age))
        return (False, age)
    return (True, age)
```

最后一个要点是main()函数，该函数请求用户的姓名和年龄，并根据用户的年龄决定该玩哪个游戏，如下所示。

```
def main():
    name = input("Hello, What's your name? ")
    valid_input = False
    while not valid_input:
        valid_input, age = validate_age(name)
    game = FrogWorld if age < 18 else WizardWorld
    environment = GameEnvironment(game(name))
    environment.play()
```

抽象工厂实现的完整代码（abstract_factory.py）如下所示。

```
class Frog:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
        print('{} the Frog encounters {} and {}!'.format(self,
        obstacle, obstacle.action()))

class Bug:
    def __str__(self):
        return 'a bug'

    def action(self):
        return 'eats it'

class FrogWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t------ Frog World -------'

    def make_character(self):
        return Frog(self.player_name)

    def make_obstacle(self):
        return Bug()

class Wizard:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
        print('{} the Wizard battles against {} and {}!'.format(self, obstacle,
        obstacle.action()))

class Ork:
    def __str__(self):
        return 'an evil ork'

    def action(self):
        return 'kills it'

class WizardWorld:
```

```
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t------ Wizard World -------'

    def make_character(self):
        return Wizard(self.player_name)

    def make_obstacle(self):
        return Ork()

class GameEnvironment:
    def __init__(self, factory):
        self.hero = factory.make_character()
        self.obstacle = factory.make_obstacle()

    def play(self):
        self.hero.interact_with(self.obstacle)

def validate_age(name):
    try:
        age = input('Welcome {}. How old are you? '.format(name))
        age = int(age)
    except ValueError as err:
        print("Age {} is invalid, please try again...".format(age))
        return (False, age)
    return (True, age)

def main():
    name = input("Hello. What's your name? ")
    valid_input = False
    while not valid_input:
        valid_input, age = validate_age(name)
    game = FrogWorld if age < 18 else WizardWorld
    environment = GameEnvironment(game(name))
    environment.play()

if __name__ == '__main__':
    main()
```

该程序的一个样例输出如下所示。

```
>>> python3 abstract_factory.py
Hello. What's your name? Nick
Welcome Nick. How old are you? 17
        ------ Frog World -------
Nick the Frog encounters a bug and eats it!
```

来尝试扩展一下这个游戏使其更完整吧。你可以随意添加障碍物、敌人以及其他任何想要的东西。

1.3　小结

本章中，我们学习了如何使用工厂方法和抽象工厂设计模式。两种模式都可以用于以下几种场景：(a)想要追踪对象的创建时，(b)想要将对象的创建与使用解耦时，(c)想要优化应用的性能和资源占用时。场景(c)在本章中并未详细说明，你也许可以将其作为一个练习。

工厂方法设计模式的实现是一个不属于任何类的单一函数，负责单一种类对象（一个形状、一个连接点或者其他对象）的创建。我们看到工厂方法是如何与玩具制造相关联的，提到Django是如何将其用于创建不同表单字段的，并讨论了其他可能的应用案例。作为示例，我们实现了一个工厂方法，提供了访问XML和JSON文件的能力。

抽象工厂设计模式的实现是同属于单个类的许多个工厂方法用于创建一系列种类的相关对象（一辆车的部件、一个游戏的环境，或者其他对象）。我们提到抽象工厂如何与汽车制造业相关联，Django程序包`django_factory`是如何利用抽象工厂创建干净的测试用例，并学习了抽象工厂的应用案例。作为抽象工厂实现的示例，我们完成了一个迷你游戏，演示了如何在单个类中使用多个相关工厂。

接下来在第2章中，我们将谈论建造者模式，它是另一种创建型模式，可用于细粒度控制复杂对象的创建过程。

第 2 章

建造者模式

想象一下，我们想要创建一个由多个部分构成的对象，而且它的构成需要一步接一步地完成。只有当各个部分都创建好，这个对象才算是完整的。这正是*建造者设计模式*（Builder design pattern）的用武之地。建造者模式将一个复杂对象的构造过程与其表现分离，这样，同一个构造过程可用于创建多个不同的表现（请参考［GOF95，第110页］和网页［t.cn/RqBBKyf］）。

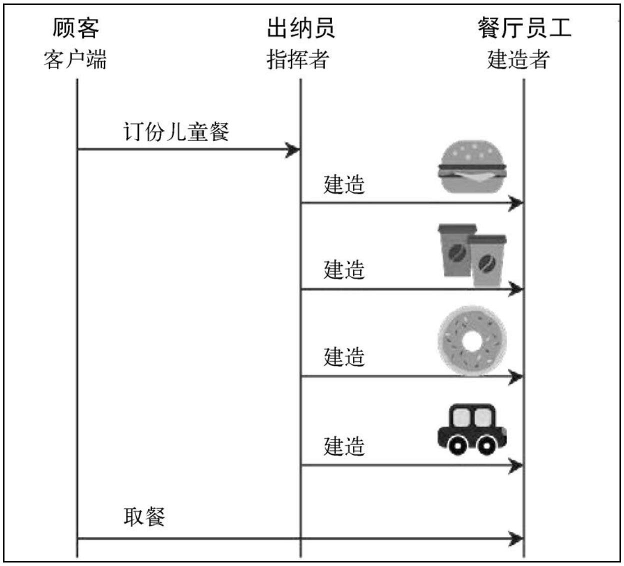
我们来看个实际的例子，这可能有助于理解建造者模式的目的。假设我们想要创建一个HTML页面生成器，HTML页面的基本结构（构造组件）通常是一样的：以`<html>`开始`</html>`结束，在HTML部分中有`<head>`和`</head>`元素，在head部分中又有`<title>`和`</title>`元素，等等；但页面在表现上可以不同。每个页面有自己的页面标题、文本标题以及不同的`<body>`内容。此外，页面通常是经过多个步骤创建完成的：有一个函数添加页面标题，另一个添加主文本标题，还有一个添加页脚，等等。仅当一个页面的结构全部完成后，才能使用一个最终的渲染函数将该页面展示在客户端。我们甚至可以更进一步扩展这个HTML生成器，让它可以生成一些完全不同的HTML页面。一个页面可能包含表格，另一个页面可能包含图像库，还有一个页面包含联系表单，等等。

HTML页面生成问题可以使用建造者模式来解决。该模式中，有两个参与者：*建造者*（builder）和*指挥者*（director）。建造者负责创建复杂对象的各个组成部分。在HTML例子中，这些组成部分是页面标题、文本标题、内容主体及页脚。指挥者使用一个建造者实例控制建造的过程。对于HTML示例，这是指调用建造者的函数设置页面标题、文本标题等。使用不同的建造者实例让我们可以创建不同的HTML页面，而无需变更指挥者的代码。

2.1　现实生活的例子

快餐店使用的就是建造者设计模式。即使存在多种汉堡包（经典款、奶酪汉堡包等）和不同包装（小盒子、中等大小盒子等），准备一个汉堡包及打包（盒子或纸袋）的流程都是相同的。经典款汉堡包和奶酪汉堡包之间的区别在于表现，而不是建造过程。指挥者是出纳员，将需要准备什么餐品的指令传达给工作人员，建造者是工作人员中的个体，关注具体的顺序。下图由www.sourcemaking.com提供，展示了统一建模语言（UML）的流程图，说明当一个儿童套餐下单

时，发生在顾客（客户端）、出纳员（指挥者）、工作人员（建造者）之间的信息交流（请参考网页［t.cn/RqBBKyf］）。



2.2　软件的例子

本章一开始提到的HTML例子，在django-widgy中得到了实际应用。django-widgy是一个Django的第三方树编辑器扩展，可用作内容管理系统（Content Management System，CMS）。它包含一个网页构建器，用来创建具有不同布局的HTML页面（请参考网页［t.cn/RqBBOBE］）。

django-query-builder是另一个基于建造者模式的Django第三方扩展库，该扩展库可用于动态地构建SQL查询。使用它，我们能够控制一个查询的方方面面，并能创建不同种类的查询，从简单的到非常复杂的都可以（请参考网页［t.cn/RqBBTrA］）。

2.3　应用案例

如果我们知道一个对象必须经过多个步骤来创建，并且要求同一个构造过程可以产生不同的表现，就可以使用建造者模式。这种需求存在于许多应用中，例如页面生成器（本章提到的HTML页面生成器之类）、文档转换器（请参考［GOF95，第110页］）以及用户界面（User Interface，UI）表单创建工具（请参考网页［t.cn/RqBBDwb］）。

有些资料提到建造者模式也可用于解决可伸缩构造函数问题（请参考网页［t.cn/RGcGaLD］）。当我们为支持不同的对象创建方式而不得不创建一个新的构造函数时，可伸缩构造函数问题就发

生了，这种情况最终产生许多构造函数和长长的形参列表，难以管理。Stack Overflow网站上列出了一个可伸缩构造函数的例子（请参考网页［t.cn/RqBrwzP］)。幸运的是，这个问题在Python中并不存在，因为至少有以下两种方式可以解决这个问题。

- 使用命名形参（请参考网页［t.cn/RqBrUyV］）
- 使用实参列表展开（请参考网页［t.cn/RyHhfg3)］）

在这一点上，建造者模式和工厂模式的差别并不太明确。主要的区别在于工厂模式以单个步骤创建对象，而建造者模式以多个步骤创建对象，并且几乎始终会使用一个指挥者。一些有针对性的建造者模式实现并未使用指挥者，如Java的StringBuilder，但这只是例外。

另一个区别是，在工厂模式下，会立即返回一个创建好的对象；而在建造者模式下，仅在需要时客户端代码才显式地请求指挥者返回最终的对象（请参考［GOF95，第113页］和网页［t.cn/RqBBKyf］)。

新电脑类比的例子也许有助于区分建造者模式和工厂模式。假设你想购买一台新电脑，如果决定购买一台特定的预配置的电脑型号，例如，最新的苹果1.4GHz Mac mini，则是在使用工厂模式。所有硬件的规格都已经由制造商预先确定，制造商不用向你咨询就知道自己该做些什么，它们通常接收的仅仅是单条指令。在代码级别上，看起来是下面这样的（apple-factory.py）。

```
MINI14 = '1.4GHz Mac mini'

class AppleFactory:
    class MacMini14:
        def __init__(self):
            self.memory = 4 # 单位为GB
            self.hdd = 500 # 单位为GB
            self.gpu = 'Intel HD Graphics 5000'

        def __str__(self):
            info = ('Model: {}'.format(MINI14),
                    'Memory: {}GB'.format(self.memory),
                    'Hard Disk: {}GB'.format(self.hdd),
                    'Graphics Card: {}'.format(self.gpu))
            return '\n'.join(info)

    def build_computer(self, model):
        if (model == MINI14):
            return self.MacMini14()
        else:
            print("I dont't know how to build {}".format(model))

if __name__ == '__main__':
    afac = AppleFactory()
    mac_mini = afac.build_computer(MINI14)
    print(mac_mini)
```

这里嵌套了MacMini14类。这是禁止直接实例化一个类的简洁方式。

另一个选择是购买一台定制的PC。假若这样，使用的即是建造者模式。你是指挥者，向制造商（建造者）提供指令说明心中理想的电脑规格。在代码方面，看起来是下面这样的（computer-builder.py）。

```
class Computer:
    def __init__(self, serial_number):
        self.serial = serial_number
        self.memory = None      # 单位为GB
        self.hdd = None         # 单位为GB
        self.gpu = None

    def __str__(self):
        info = ('Memory: {}GB'.format(self.memory),
                'Hard Disk: {}GB'.format(self.hdd),
                'Graphics Card: {}'.format(self.gpu))
        return '\n'.join(info)

class ComputerBuilder:
    def __init__(self):
        self.computer = Computer('AG23385193')

    def configure_memory(self, amount):
        self.computer.memory = amount

    def configure_hdd(self, amount):
        self.computer.hdd = amount

    def configure_gpu(self, gpu_model):
        self.computer.gpu = gpu_model

class HardwareEngineer:
    def __init__(self):
        self.builder = None

    def construct_computer(self, memory, hdd, gpu):
        self.builder = ComputerBuilder()①
        [step for step in (self.builder.configure_memory(memory),
                           self.builder.configure_hdd(hdd),
                           self.builder.configure_gpu(gpu))]

    @property
    def computer(self):
        return self.builder.computer

def main():
    engineer = HardwareEngineer()
```

① 下面这句没必要使用列表推导式。——译者注

```
    engineer.construct_computer(hdd=500, memory=8, gpu='GeForce GTX 650 Ti')
    computer = engineer.computer
    print(computer)

if __name__ == '__main__':
    main()
```

基本的变化是引入了一个建造者ComputerBuilder、一个指挥者HardwareEngineer以及一步接一步装配一台电脑的过程，这样现在就支持不同的配置了（注意，memory、hdd及gpu是形参，并未预先设置）。如果我们想要支持平板电脑的装配，那又需要做些什么呢？作为练习来实现它吧。

你或许还想将每台电脑的serial_number变得都不一样，因为现在所有电脑的序列号都相同，这是不符合实际情况的。

2.4　实现

让我们来看看如何使用建造者设计模式实现一个比萨订购的应用。比萨的例子特别有意思，因为准备好一个比萨需经过多步操作，且这些操作要遵从特定顺序。要添加调味料，你得先准备生面团。要添加配料，你得先添加调味料。并且只有当生面团上放了调味料和配料之后才能开始烤比萨。此外，每个比萨通常要求的烘培时间都不一样，依赖于生面团的厚度和使用的配料。

先导入要求的模块，声明一些Enum参数（请参考网页［t.cn/RqBrIpz］）以及一个在应用中会使用多次的常量。常量STEP_DELAY用于在准备一个比萨的不同步骤（准备生面团、添加调味料等）之间添加时间延迟，如下所示。

```
from enum import Enum

PizzaProgress = Enum('PizzaProgress', 'queued preparation baking ready')
PizzaDough = Enum('PizzaDough', 'thin thick')
PizzaSauce = Enum('PizzaSauce', 'tomato creme_fraiche')
PizzaTopping = Enum('PizzaTopping', 'mozzarella double_mozzarella bacon ham mushrooms
red_onion oregano')
STEP_DELAY = 3                  # 考虑是示例，所以单位为秒
```

最终的产品是一个比萨，由Pizza类描述。若使用建造者模式，则最终产品（类）并没有多少职责，因为它不支持直接实例化。建造者会创建一个最终产品的实例，并确保这个实例完全准备好。这就是Pizza类这么短小的缘由。它只是将所有数据初始化为合理的默认值，唯一的例外是方法prepare_dough()。将prepare_dough方法定义在Pizza类而不是建造者中，是考虑到以下两点。

- 为了澄清一点，就是虽然最终产品类通常会最小化，但这并不意味着绝不应该给它分配任何职责
- 为了通过组合提高代码复用（请参考［GOF95，第32页］）

```
class Pizza:
    def __init__(self, name):
        self.name = name
        self.dough = None
        self.sauce = None
        self.topping = []

    def __str__(self):
        return self.name

    def prepare_dough(self, dough):
        self.dough = dough
        print('preparing the {} dough of your {}...'.format(self.dough.name, self))
        time.sleep(STEP_DELAY)
        print('done with the {} dough'.format(self.dough.name))
```

该应用中有两个建造者：一个制作玛格丽特比萨（`MargaritaBudiler`），另一个制作奶油熏肉比萨（`CreamyBaconBuilder`）。每个建造者都创建一个`Pizza`实例，并包含遵从比萨制作流程的方法：`prepare_dough()`、`add_sauce`、`add_topping()`和`bake()`。准确来说，其中的`prepare_dough`只是对`Pizza`类中`prepare_dough()`方法的一层封装。注意每个建造者是如何处理所有比萨相关细节的。例如，玛格丽特比萨的配料是双层马苏里拉奶酪（mozzarella）和牛至（oregano），而奶油熏肉比萨的配料是马苏里拉奶酪（mozzarella）、熏肉（bacon）、火腿（ham）、蘑菇（mushrooms）、紫洋葱（red onion）和牛至（oregano），如下面的代码所示。

```
class MargaritaBuilder:
    def __init__(self):
        self.pizza = Pizza('margarita')
        self.progress = PizzaProgress.queued
        self.baking_time = 5      # 考虑是示例，单位为秒

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough.thin)

    def add_sauce(self):
        print('adding the tomato sauce to your margarita...')
        self.pizza.sauce = PizzaSauce.tomato
        time.sleep(STEP_DELAY)
        print('done with the tomato sauce')

    def add_topping(self):
        print('adding the topping (double mozzarella, oregano) to your margarita')
        self.pizza.topping.append([i for i in (PizzaTopping.double_mozzarella,
        PizzaTopping.oregano)])
        time.sleep(STEP_DELAY)
        print('done with the topping (double mozzarella, oregano)')

     def bake(self):
         self.progress = PizzaProgress.baking
         print('baking your margarita for {}
         seconds'.format(self.baking_time))
         time.sleep(self.baking_time)
```

```
        self.progress = PizzaProgress.ready
        print('your margarita is ready')

class CreamyBaconBuilder:
    def __init__(self):
        self.pizza = Pizza('creamy bacon')
        self.progress = PizzaProgress.queued
        self.baking_time = 7      # 考虑是示例，单位为秒

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough.thick)

    def add_sauce(self):
        print('adding the crème fraîche sauce to your creamy bacon')

        self.pizza.sauce = PizzaSauce.creme_fraiche
        time.sleep(STEP_DELAY)

        print('done with the crème fraîche sauce')

    def add_topping(self):
        print('adding the topping (mozzarella, bacon, ham,
        mushrooms, red onion, oregano) to your creamy bacon')
        self.pizza.topping.append([t for t in (PizzaTopping.mozzarella,
        PizzaTopping.bacon,
        PizzaTopping.ham,PizzaTopping.mushrooms,
        PizzaTopping.red_onion, PizzaTopping.oregano)])
        time.sleep(STEP_DELAY)
        print('done with the topping (mozzarella, bacon, ham, mushrooms, red onion,
        oregano)')

    def bake(self):
        self.progress = PizzaProgress.baking
        print('baking your creamy bacon for {} seconds'.format(self.baking_time))
        time.sleep(self.baking_time)
        self.progress = PizzaProgress.ready
        print('your creamy bacon is ready')
```

在这个例子中，指挥者就是服务员。`Waiter`类的核心是`construct_pizza`方法，该方法接受一个建造者作为参数，并以正确的顺序执行比萨的所有准备步骤。选择恰当的建造者（甚至可以在运行时选择），无需修改指挥者（`Waiter`）的任何代码，就能制作不同的比萨。`Waiter`类还包含`pizza()`方法，会向调用者返回最终产品（准备好的比萨），如下所示。

```
class Waiter:
    def __init__(self):
        self.builder = None

    def construct_pizza(self, builder):
        self.builder = builder
        [step() for step in (builder.prepare_dough, builder.add_sauce,
        builder.add_topping, builder.bake)]

    @property
```

```
    def pizza(self):
        return self.builder.pizza
```

函数validate_style()类似于第1章中描述的validate_age()函数，用于确保用户提供有效的输入，当前案例中这个输入是映射到一个比萨建造者的字符；输入字符m表示使用MargaritaBuilder类，输入字符c则使用CreamyBaconBuilder类。这些映射关系存储在参数builder中。该函数会返回一个元组，如果输入有效，则元组的第一个元素被设置为True， 否则为False，如下所示。

```
def validate_style(builders):
    try:
        pizza_style = input('What pizza would you like, [m]argarita or [c]reamy bacon?
        ')
        builder = builders[pizza_style]()
        valid_input = True
    except KeyError as err:
        print('Sorry, only margarita (key m) and creamy bacon (key c) are available')
        return (False, None)
    return (True, builder)
```

实现的最后一部分是main()函数。main()函数实例化一个比萨建造者，然后指挥者Waiter使用比萨建造者来准备比萨。创建好的比萨可在稍后的时间点交付给客户端。

```
def main():
    builders = dict(m=MargaritaBuilder, c=CreamyBaconBuilder)
    valid_input = False
    while not valid_input:
        valid_input, builder = validate_style(builders)
    print()
    waiter = Waiter()
    waiter.construct_pizza(builder)
    pizza = waiter.pizza
    print()
    print('Enjoy your {}!'.format(pizza))
```

将所有代码片段拼接在一起，示例的完整代码（builder.py）如下所示。

```
from enum import Enum
import time

PizzaProgress = Enum('PizzaProgress', 'queued preparation baking ready')
PizzaDough = Enum('PizzaDough', 'thin thick')
PizzaSauce = Enum('PizzaSauce', 'tomato creme_fraiche')
PizzaTopping = Enum('PizzaTopping', 'mozzarella double_mozzarella bacon ham mushrooms
red_onion oregano')
STEP_DELAY = 3            # 考虑到这是示例，单位为秒

class Pizza:
    def __init__(self, name):
        self.name = name
        self.dough = None
```

```
        self.sauce = None
        self.topping = []

    def __str__(self):
        return self.name

    def prepare_dough(self, dough):
        self.dough = dough
        print('preparing the {} dough of your {}...'.format(self.dough.name, self))
        time.sleep(STEP_DELAY)
        print('done with the {} dough'.format(self.dough.name))

class MargaritaBuilder:
    def __init__(self):
        self.pizza = Pizza('margarita')
        self.progress = PizzaProgress.queued
        self.baking_time = 5        # 考虑是示例，单位为秒

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough.thin)

    def add_sauce(self):
        print('adding the tomato sauce to your margarita...')
        self.pizza.sauce = PizzaSauce.tomato
        time.sleep(STEP_DELAY)
        print('done with the tomato sauce')

    def add_topping(self):
        print('adding the topping (double mozzarella, oregano) to your margarita')
        self.pizza.topping.append([i for i in
        (PizzaTopping.double_mozzarella, PizzaTopping.oregano)])
        time.sleep(STEP_DELAY)
        print('done with the topping (double mozzarrella, oregano)')

    def bake(self):
        self.progress = PizzaProgress.baking
        print('baking your margarita for {} seconds'.format(self.baking_time))
        time.sleep(self.baking_time)
        self.progress = PizzaProgress.ready
        print('your margarita is ready')

class CreamyBaconBuilder:
    def __init__(self):
        self.pizza = Pizza('creamy bacon')
        self.progress = PizzaProgress.queued
        self.baking_time = 7        # 考虑是示例，单位为秒

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough.thick)

    def add_sauce(self):
        print('adding the crème fraîche sauce to your creamy bacon')
```

```
        self.pizza.sauce = PizzaSauce.creme_fraiche
        time.sleep(STEP_DELAY)
        print('done with the crème fraîche sauce')

    def add_topping(self):
        print('adding the topping (mozzarella, bacon, ham, mushrooms, red onion,
        oregano) to your creamy bacon')
        self.pizza.topping.append([t for t in
        (PizzaTopping.mozzarella, PizzaTopping.bacon,
        PizzaTopping.ham,PizzaTopping.mushrooms,
        PizzaTopping.red_onion, PizzaTopping.oregano)])
        time.sleep(STEP_DELAY)
        print('done with the topping (mozzarella, bacon, ham, mushrooms, red onion,
        oregano)')

    def bake(self):
        self.progress = PizzaProgress.baking
        print('baking your creamy bacon for {} seconds'.format(self.baking_time))
        time.sleep(self.baking_time)
        self.progress = PizzaProgress.ready
        print('your creamy bacon is ready')

class Waiter:
    def __init__(self):
        self.builder = None

    def construct_pizza(self, builder):
        self.builder = builder
        [step() for step in (builder.prepare_dough,
        builder.add_sauce, builder.add_topping, builder.bake)]

    @property
    def pizza(self):
        return self.builder.pizza

def validate_style(builders):
    try:
        pizza_style = input('What pizza would you like, [m]argarita or [c]reamy bacon?
        ')
        builder = builders[pizza_style]()
        valid_input = True
    except KeyError as err:
        print('Sorry, only margarita (key m) and creamy bacon (key c) are available')
        return (False, None)
    return (True, builder)

def main():
    builders = dict(m=MargaritaBuilder, c=CreamyBaconBuilder)
    valid_input = False
    while not valid_input:
        valid_input, builder = validate_style(builders)
    print()
    waiter = Waiter()
    waiter.construct_pizza(builder)
```

```
    pizza = waiter.pizza
    print()
    print('Enjoy your {}!'.format(pizza))

if __name__ == '__main__':
    main()
```

以上示例代码的样例输出如下所示。

```
>>> python3 builder.py
What pizza would you like, [m]argarita or [c]reamy bacon? r
Sorry, only margarita (key m) and creamy bacon (key c) are available
What pizza would you like, [m]argarita or [c]reamy bacon? m

preparing the thin dough of your margarita...
done with the thin dough
adding the tomato sauce to your margarita...
done with the tomato sauce
adding the topping (double mozzarella, oregano) to your margarita
done with the topping (double mozzarella, oregano)
baking your margarita for 5 seconds
your margarita is ready

Enjoy your margarita!
```

程序仅支持两种比萨类型是挺丢脸的。你自己再来实现一个夏威夷比萨建造者。权衡利弊之后考虑一下是否使用继承。看看典型夏威夷比萨的原料，再决定通过扩展哪个类来实现：`MargaritaBuilder`或`CreamyBaconBuilder`？或许两者皆扩展（请参考网页［t.cn/RqBr-XK5］）？

在*Effective Java* (2nd edition)一书中，Joshua Bloch描述了一种有趣的建造者模式变体，这种变体会链式地调用建造者方法，通过将建造者本身定义为内部类并从其每个设置器方法返回自身来实现。方法`build()`返回最终的对象。这个模式被称为流利的建造者。以下是其Python实现，由本书的一位评审人友情提供。

```
class Pizza:
    def __init__(self, builder):
        self.garlic = builder.garlic
        self.extra_cheese  = builder.extra_cheese

    def __str__(self):
        garlic = 'yes' if self.garlic else 'no'
        cheese = 'yes' if self.extra_cheese else 'no'
        info = ('Garlic: {}'.format(garlic), 'Extra cheese: {}'.format(cheese))
        return '\n'.join(info)

    class PizzaBuilder:
        def __init__(self):
            self.extra_cheese = False
            self.garlic = False
```

```
        def add_garlic(self):
            self.garlic = True
            return self

        def add_extra_cheese(self):
            self.extra_cheese = True
            return self

        def build(self):
            return Pizza(self)

if __name__ == '__main__':
    pizza = Pizza.PizzaBuilder().add_garlic().add_extra_cheese().build()
    print(pizza)
```

你可以尝试一下把流利的建造者模式应用到比萨的例子。哪个版本你更喜欢？每个版本的优势和劣势又是什么？

2.5 小结

本章中，我们学习了如何使用建造者设计模式。可以在工厂模式（工厂方法或抽象工厂）不适用的一些场景中使用建造者模式创建对象。在以下几种情况下，与工厂模式相比，建造者模式是更好的选择。

- 想要创建一个复杂对象（对象由多个部分构成，且对象的创建要经过多个不同的步骤，这些步骤也许还需遵从特定的顺序）
- 要求一个对象能有不同的表现，并希望将对象的构造与表现解耦
- 想要在某个时间点创建对象，但在稍后的时间点再访问

我们看到了快餐店如何将建造者模式用于准备食物，两个第三方Django扩展包（django-widgy和django-query-builder）各自如何使用建造者模式来生成HTML页面和动态的SQL查询。我们重点学习了建造者模式与工厂模式之间的区别，通过对预先配置（工厂）电脑与客户定制（建造者）电脑进行订单类比来理清这两种设计模式。

在实现部分，我们学习了如何创建一个比萨订购应用，该应用能处理比萨准备过程的步骤依赖。本章推荐了很多有趣的练习题，包括实现一个流利的建造者模式。

第3章将学习本书涉及的最后一个创建型设计模式，也就是原型模式，该模式用于克隆对象。

第 3 章 原型模式

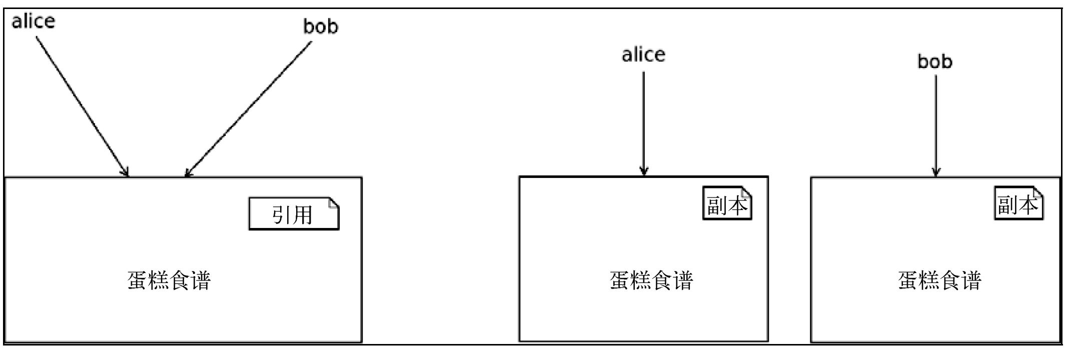
3

有时，我们需要原原本本地为对象创建一个副本。举例来说，假设你想创建一个应用来存储、分享、编辑（比如，修改、添加注释及删除）食谱。用户Bob找到一份蛋糕食谱，在做了一些改变后，觉得自己做的蛋糕非常美味，想要与朋友Alice分享这个食谱。但是该如何分享食谱呢？如果在与Alice分享之后，Bob想对食谱做进一步的试验，Alice手里的食谱也能跟着变化吗？Bob能够持有蛋糕食谱的两个副本吗？对蛋糕食谱进行的试验性变更不应该对原本美味蛋糕的食谱造成影响。

这样的问题可以通过让用户对同一份食谱持有多个独立的副本来解决。每个副本被称为一个克隆，是某个时间点原有对象的一个完全副本。这里时间是一个重要因素。因为它会影响克隆所包含的内容。例如，如果Bob在对蛋糕食谱做改进以臻完美之前就与Alice分享了，那么Alice就绝不可能像Bob那样烘烤出自己的美味蛋糕，只能按照Bob原来找到的食谱烘烤蛋糕。

注意引用与副本之间的区别。如果Bob和Alice持有的是同一个蛋糕食谱对象的两个引用，那么Bob对食谱做的任何改变，对于Alice的食谱版本都是可见的，反之亦然。我们想要的是Bob和Alice各自持有自己的副本，这样他们可以各自做变更而不会影响对方的食谱。实际上Bob需要蛋糕食谱的两个副本：美味版本和试验版本。

下图展示了引用与副本之间的区别。



左侧是两个引用。Bob和Alice参考的是同一个食谱，这本质上意味着两者共享食谱，并且所

有变更两人皆可见。右侧是同一个食谱的两个不同副本，这样允许各自独立地变更，Alice做的改变不会影响Bob做的改变，反之亦然。

原型设计模式（Prototype design pattern）帮助我们创建对象的克隆，其最简单的形式就是一个clone()函数，接受一个对象作为输入参数，返回输入对象的一个副本。在Python中，这可以使用copy.deepcopy()函数来完成。来看一个例子，下面的代码中（文件clone.py）有两个类，A和B。A是父类，B是衍生类/子类。在主程序部分，我们创建一个类B的实例b，并使用deepcopy()创建b的一个克隆c。结果是所有成员都被复制到了克隆c，以下是代码演示。作为一个有趣的练习，你也可以尝试在对象的组合形式上使用deepcopy()。


```
import copy

class A:
    def __init__(self):
        self.x = 18
        self.msg = 'Hello'

class B(A):
    def __init__(self):
        A.__init__(self)
        self.y = 34

    def __str__(self):
        return '{}, {}, {}'.format(self.x, self.msg, self.y)

if __name__ == '__main__':
    b = B()
    c = copy.deepcopy(b)
    print([str(i) for i in (b, c)])
    print([i for i in (b, c)])
```

在我的电脑上执行clone.py，得到以下输出。

```
>>> python3 clone.py
['18, Hello, 34', '18, Hello, 34']
[<__main__.B object at 0x7f92dac33240>, <__main__.B object at 0x7f92dac33208>]
```

虽然你得到的第二行输出很可能与我的不一样，但重要的是注意两个对象位于两个不同的内存地址（输出中的0x...部分）。这意味着两个对象是两个独立的副本。

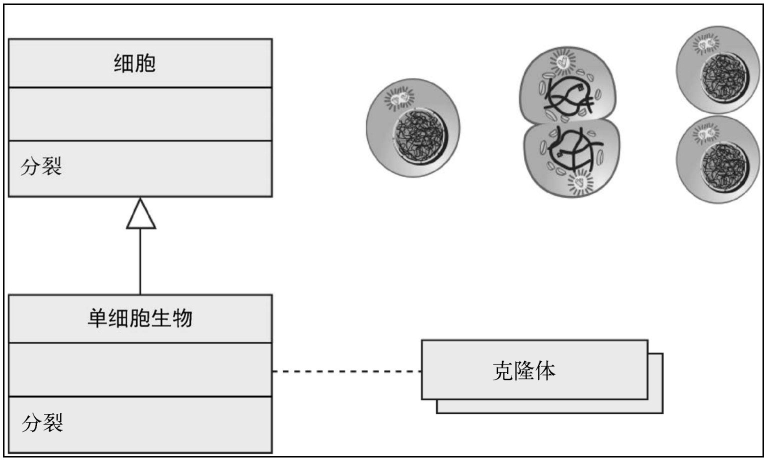
在本章稍后的3.4节中，我们将看到为了保持一个被克隆对象的注册表，如何将copy.deepcopy与封装在某个类中的一些额外的样板代码一起使用。

3.1 现实生活的例子

原型设计模式无非就是克隆一个对象。有丝分裂，即细胞分裂的过程，是生物克隆的一个例子。在这个过程中，细胞核分裂产生两个新的细胞核，其中每个都有与原来细胞完全相同的染色

体和DNA内容（请参考网页［t.cn/RqBrOuM］）。

下图由www.sourcemaking.com提供，展示了细胞有丝分裂的一个例子（请参考网页［t.cn/RqBrYa8］）。



另一个著名的（人工）克隆例子是多利羊（请参考网页［t.cn/RGod5lh］）。

3.2 软件的例子

很多Python应用都使用了原型模式（请参考网页［t.cn/RqBruae］），但几乎都不称之为原型模式，因为对象克隆是编程语言的一个内置特性。

可视化工具套件（Visualization Toolkit，VTK）（请参考网页［t.cn/hCDIf］）是原型模式的一个应用。VTK是一个开源的跨平台系统，用于三维计算机图形/图片处理以及可视化。VTK使用原型模式来创建几何元素（比如，点、线、六面体等，请参考网页［t.cn/RqBrecy］）的克隆。

另一个使用原型模式的项目是music21。根据该项目页面所述，"music21是一组工具，帮助学者和其他积极的听众快速简便地得到音乐相关问题的答案"（请参考网页［t.cn/RGK8f1V］）。music21工具套件使用原型模式来复制音符和总谱（请参考网页［t.cn/RqBdhGK］）。

3.3 应用案例

当我们已有一个对象，并希望创建该对象的一个完整副本时，原型模式就派上用场了。在我们知道对象的某些部分会被变更但又希望保持原有对象不变之时，通常需要对象的一个副本。在这样的案例中，重新创建原有对象是没有意义的（请参考网页［t.cn/RqBrOuM］）。

另一个案例是，当我们想复制一个复杂对象时，使用原型模式会很方便。对于复制复杂对象，我们可以将对象当作是从数据库中获取的，并引用其他一些也是从数据库中获取的对象。若通过多次重复查询数据来创建一个对象，则要做很多工作。在这种场景下使用原型模式要方便得多。

至此，我们仅涉及了引用与副本的问题，而副本又可以进一步分为深副本与浅副本。深副本就是我们在本章中到目前为止所看到的：原始对象的所有数据都被简单地复制到克隆对象中，没有例外。浅副本则依赖引用。我们可以引入数据共享和写时复制一类的技术来优化性能（例如，减小克隆对象的创建时间）和内存使用。如果可用资源有限（例如，嵌入式系统）或性能至关重要（例如，高性能计算），那么使用浅副本可能更佳。

在Python中，可以使用`copy.copy()`函数进行浅复制。以下内容引用自Python官方文档，说明了浅副本（`copy.copy()`）和深副本（`copy.deepcopy()`）之间的区别（请参考网页[t.cn/RqBdSj1]）。

- 浅副本构造一个新的复合对象后，（会尽可能地）将在原始对象中找到的对象的引用插入新对象中。
- 深副本构造一个新的复合对象后，会递归地将在原始对象中找到的对象的副本插入新对象中。

你能想到什么使用浅副本比深副本更好的例子吗？

3.4　实现

编程方面的书籍历经多个版本的情况并不多见。例如，Kernighan和Ritchie编著的C编程经典教材《C程序设计语言》（*The C Programming Language*）历经两个版本。第一版1978年出版，那时C语言还没被标准化。该书第二版10年后才出版，涵盖C语言标准（ANSI）版本。这两个版本之间的区别是什么？列举几个：价格、长度（页数）以及出版日期。但也有很多相似之处：作者、出版商以及描述该书的标签/关键词都是完全一样的。这表明从头创建一版新书并不总是最佳方式。如果知道两个版本之间的诸多相似之处，则可以先克隆一份，然后仅修改新版本与旧版本之间的不同之处。

来看看可以如何使用原型模式创建一个展示图书信息的应用。我们以一本书的描述开始。除了常规的初始化之外，`Book`类展示了一种有趣的技术可避免可伸缩构造器问题。在`__init__()`方法中，仅有三个形参是固定的：`name`、`authors`和`price`，但是使用`rest`变长列表，调用者能以关键词的形式（名称=值）传入更多的参数。`self.__dict__.update(rest)`一行将`rest`的内容添加到`Book`类的内部字典中，成为它的一部分。

但这里有个问题。我们并不知道所有被添加参数的名称，但又需要访问内部字典将这些参数应用到`__str__()`中，并且字典的内容并不遵循任何特定的顺序，所以使用一个`OrderedDict`

来强制元素有序，否则，每次程序执行都会产生不同的输出。当然，你不应该把我的话当作理所当然。作为一个练习，尝试删除使用的OrderedDict和sorted()，运行一下示例代码看看我说得对不对。

```
class Book:
    def __init__(self, name, authors, price, **rest):
        '''rest的例子有：出版商、长度、 标签、出版日期'''
        self.name = name
        self.authors = authors
        self.price = price        # 单位为美元
        self.__dict__.update(rest)

    def __str__(self):
        mylist = []
        ordered = OrderedDict(sorted(self.__dict__.items()))
        for i in ordered.keys():
            mylist.append('{}: {}'.format(i, ordered[i]))
            if i == 'price':
                mylist.append('$')
            mylist.append('\n')
        return ''.join(mylist)
```

Prototype类实现了原型设计模式。Prototype类的核心是clone()方法，该方法使用我们熟悉的copy.deepcopy()函数来完成真正的克隆工作。但Prototype类在支持克隆之外做了一点更多的事情，它包含了方法register()和unregister()，这两个方法用于在一个字典中追踪被克隆的对象。注意这仅是一个方便之举，并非必需。

此外，clone()方法和Book类中的__str__使用了相同的技巧，但这次是因为别的原因。使用变长列表attr，我们可以仅传递那些在克隆一个对象时真正需要变更的属性变量，如下所示。

```
class Prototype:
    def __init__(self):
        self.objects = dict()

    def register(self, identifier, obj):
        self.objects[identifier] = obj

    def unregister(self, identifier):
        del self.objects[identifier]

    def clone(self, identifier, **attr):
        found = self.objects.get(identifier)
        if not found:
            raise ValueError('Incorrect object identifier: {}'.format(identifier))
        obj = copy.deepcopy(found)
        obj.__dict__.update(attr)
        return obj
```

main()函数以实践的方式展示了本节开头提到的《C程序设计语言》一书克隆的例子。克隆该书的第一个版本来创建第二个版本，我们仅需要传递已有参数中被修改参数的值，但也可以传

递额外的参数。在这个案例中，edition就是一个新参数，在书的第一个版本中并不需要，但对于克隆版本却是很有用的信息。

```
def main():
    b1 = Book('The C Programming Language', ('Brian W. Kernighan', 'Dennis M.Ritchie'),
    price=118, publisher='Prentice Hall', length=228, publication_date='1978-02-22',
    tags=('C', 'programming', 'algorithms', 'data structures'))

    prototype = Prototype()
    cid = 'k&r-first'
    prototype.register(cid, b1)
    b2 = prototype.clone(cid, name='The C Programming Language(ANSI)', price=48.99,
    length=274, publication_date='1988-04-01', edition=2)

    for i in (b1, b2):
        print(i)
    print("ID b1 : {} != ID b2 : {}".format(id(b1), id(b2)))
```

注意，代码中使用了函数id()返回对象的内存地址。当使用深副本来克隆一个对象时，克隆对象的内存地址必定与原始对象的内存地址不一样。

文件prototype.py的完整内容如下所示。

```
import copy
from collections import OrderedDict

class Book:
    def __init__(self, name, authors, price, **rest):
        '''rest的例子有：出版商、长度、标签、出版日期'''
        self.name = name
        self.authors = authors
        self.price = price        # 单位为美元
        self.__dict__.update(rest)

    def __str__(self):
        mylist = []
        ordered = OrderedDict(sorted(self.__dict__.items()))
        for i in ordered.keys():
            mylist.append('{}: {}'.format(i, ordered[i]))
            if i == 'price':
                mylist.append('$')
            mylist.append('\n')
        return ''.join(mylist)

class Prototype:
    def __init__(self):
        self.objects = dict()

    def register(self, identifier, obj):
        self.objects[identifier] = obj

    def unregister(self, identifier):
```

```
        del self.objects[identifier]

    def clone(self, identifier, **attr):
        found = self.objects.get(identifier)
        if not found:
            raise ValueError('Incorrect object identifier: {}'.format(identifier))
        obj = copy.deepcopy(found)
        obj.__dict__.update(attr)
        return obj

def main():
    b1 = Book('The C Programming Language', ('Brian W. Kernighan', 'Dennis M.Ritchie'),
    price=118, publisher='Prentice Hall', length=228, publication_date='1978-02-22',
    tags=('C', 'programming', 'algorithms', 'data structures'))

    prototype = Prototype()
    cid = 'k&r-first'
    prototype.register(cid, b1)
    b2 = prototype.clone(cid, name='The C Programming Language(ANSI)', price=48.99,
length=274, publication_date='1988-04-01', edition=2)

    for i in (b1, b2):
        print(i)
    print("ID b1 : {} != ID b2 : {}".format(id(b1), id(b2)))

if __name__ == '__main__':
    main()
```

`id()`的输出依赖计算机当前的内存分配情况，该输出在每次执行这个程序时都应该是不一样的。无论实际的内存地址是多少，原始对象与克隆对象的内存地址都绝无可能相同。

在我的电脑中执行这个程序，其输出如下所示。

```
>>> python3 prototype.py
authors: ('Brian W. Kernighan', 'Dennis M. Ritchie')
length: 228
name: The C Programming Language
price: 118$
publication_date: 1978-02-22
publisher: Prentice Hall
tags: ('C', 'programming', 'algorithms', 'data structures')

authors: ('Brian W. Kernighan', 'Dennis M. Ritchie')
edition: 2
length: 274
name: The C Programming Language (ANSI)
price: 48.99$
publication_date: 1988-04-01
publisher: Prentice Hall
tags: ('C', 'programming', 'algorithms', 'data structures')

ID b1 : 140004970829304 != ID b2 : 140004970829472
```

原型模式确实按预期生效了。《C程序设计语言》的第二版复用了第一版设置的所有信息，所有我们定义的不同之处仅应用于第二版，第一版不受影响。看到`id()`函数的输出显示两个内存地址不相同，我们就更加确信了。

作为练习，你可以提出自己的原型模式的例子。以下是一些想法。

- 本章提到的食谱例子
- 本章提到的从数据库获取数据填充对象的例子
- 复制一张图片，这样你可以做些自己的修改而不会影响原来的图片

3.5　小结

在本章中，我们学习了如何使用原型设计模式。原型模式用于创建对象的完全副本。确切地说，创建一个对象的副本可以指代以下两件事情。

- 当创建一个浅副本时，副本依赖引用
- 当创建一个深副本时，副本复制所有东西

第一种情况中，我们关注提升应用性能和优化内存使用，在对象之间引入数据共享，但需要小心地修改数据，因为所有变更对所有副本都是可见的。浅副本在本章中没有过多介绍，但也许你会想试验一下。

第二种情况中，我们希望能够对一个副本进行更改而不会影响其他对象。对于我们之前看到的蛋糕食谱示例这类案例，这一特性是很有用的。这里不会进行数据共享，所以需要关注因对象克隆而引入的资源耗用问题。

我们展示了一个深副本的简单示例，在Python中，深副本使用`copy.deepcopy()`函数来完成。我们也提到一些现实生活中发现的克隆例子，并着重讲述了有丝分裂的例子。

许多软件项目都使用了原型模式，但因为在Python中这是一个内置特性，所以并不称之为原型模式。这些软件项目包括VTK（它使用原型模式创建几何学元素的克隆）和music21（它使用原型模式复制乐谱和音符）。

最后，我们讨论了原型模式的应用案例，并实现一个程序以支持克隆书籍对象。这样在新版本中所有信息无需改变即可复用，并且同时可以更新变更的信息，并添加新的信息。

原型模式是本书涉及的最后一个创建型设计模式。第4章将介绍适配器模式，它是一种结构型设计模式，可用于实现不兼容软件之间的接口兼容。

Part 2 第二部分

结构型模式

本部分内容

第 4 章

适配器模式

结构型设计模式处理一个系统中不同实体（比如，类和对象）之间的关系，关注的是提供一种简单的对象组合方式来创造新功能（请参考［GOF95，第155页］和网页［t.cn/RqBdWzD］）。

适配器模式（Adapter pattern）是一种结构型设计模式，帮助我们实现两个不兼容接口之间的兼容。首先，解释一下不兼容接口的真正含义。如果我们希望把一个老组件用于一个新系统中，或者把一个新组件用于一个老系统中，不对代码进行任何修改两者就能够通信的情况很少见。但又并非总是能修改代码，或因为我们无法访问这些代码（例如，组件以外部库的方式提供），或因为修改代码本身就不切实际。在这些情况下，我们可以编写一个额外的代码层，该代码层包含让两个接口之间能够通信需要进行的所有修改。这个代码层就叫适配器。

电子商务系统是这方面众所周知的例子。假设我们使用的一个电子商务系统中包含一个`calculate_total(order)`函数。这个函数计算一个订单的总金额，但货币单位为丹麦克朗（Danish Kroner，DKK）。顾客让我们支持更多的流行货币，比如美元（United States Dollar，USD）和欧元（Euro，EUR），这是很合理的要求。如果我们拥有系统的源代码，那么可以扩展系统，方法是添加一些新函数，将金额从DKK转换成USD，或者从DKK转换成EUR。但是如果应用仅以外部库的方式提供，我们无法访问其源代码，那又该怎么办呢？在这种情况下，我们仍然可以使用这个外部库（例如，调用它的方法），但无法修改/扩展它。解决方案是编写一个包装器（又名适配器）将数据从给定的DKK格式转换成期望的USD或EUR格式。

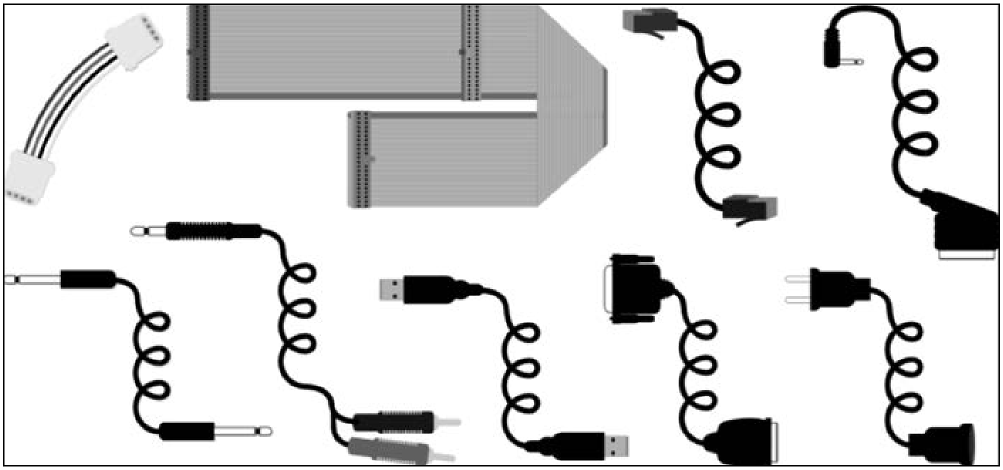
适配器模式并不仅仅对数据转换有用。通常来说，如果你想使用一个接口，期望它是`function_a()`，但仅有`function_b()`可用，那么可以使用一个适配器把`function_b()`转换（适配）成`function_a()`（请参考［Eckel08，第207页］和网页［t.cn/RqBdTCD］）。不仅对于函数可以这样做，对于函数参数也可以如此。其中一个例子是，有一个函数要求参数x、y、z，但你手头只有一个带参数x、y的函数。在4.4节我们将看到如何使用适配器模式。

4.1 现实生活的例子

也许我们所有人每天都在使用适配器模式，只不过是硬件上的，而不是软件上的。如果你有

一部智能手机或者一台平板电脑，在想把它（比如，iPhone手机的闪电接口）连接到你的电脑时，就需要使用一个USB适配器。如果你从大多数欧洲国家到英国旅行，在为你的笔记本电脑充电时，需要使用一个插头适配器。如果你从欧洲到美国旅行，同样如此；反之亦然。适配器无处不在！

下图展示了硬件适配器的若干例子（请参考网页［t.cn/RqBdTCD］），经sourcemaking.com允许使用。



4.2 软件的例子

Grok是一个Python框架，运行在Zope 3之上，专注于敏捷开发。Grok框架使用适配器，让已有对象无需变更就能符合指定API的标准（请参考网页［t.cn/RqBd1gM］）。

Python第三方包Traits也使用了适配器模式，将没有实现某个指定接口（或一组接口）的对象转换成实现了接口的对象（请参考网页［t.cn/RqBdg28］）。

4.3 应用案例

在某个产品制造出来之后，需要应对新的需求之时，如果希望其仍然有效，则可以使用适配器模式（请参考网页［t.cn/RqBdTCD］）。通常两个不兼容接口中的一个是他方的或者是老旧的。如果一个接口是他方的，就意味着我们无法访问其源代码。如果是老旧的，那么对其重构通常是不切实际的。更进一步，我们可以说修改一个老旧组件的实现以满足我们的需求，不仅是不切实际的，而且也违反了开放/封闭原则（请参考网页［t.cn/RqBdFAO］）。开放/封闭原则（open/close principle）是面向对象设计的基本原则之一（SOLID中的O），声明一个软件实体应该对扩展是开放的，对修改则是封闭的。本质上这意味着我们应该无需修改一个软件实体的源代码就能扩展其

行为。适配器模式遵从开放/封闭原则（请参考网页［t.cn/RqBghbH］）。

因此，在某个产品制造出来之后，需要应对新的需求之时，如果希望其仍然有效，使用适配器是一种更好的方式，原因如下所示。

- 不要求访问他方接口的源代码
- 不违反开放/封闭原则

4.4　实现

使用Python实现适配器设计模式有多种方法（请参考［Eckel08，第207页］。Bruce Eckel演示的所有技巧都是使用继承，但是Python提供了一种替代方案，在我看来，这种实现适配器的方式更地道一些。你应该已熟悉这一替代技巧，因为它使用了类的内部字典，在第3章中我们就看到了如何使用类的内部字典。

先来看看"我们有什么"部分。我们的应用有一个Computer类，用来显示一台计算机的基本信息。这一例子中的所有类，包括Computer类，都非常简单，因为我们希望关注适配器模式，而不是如何尽可能完善一个类。

```
class Computer:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return 'the {} computer'.format(self.name)

    def execute(self):
        return 'executes a program'
```

在这里，execute方法是计算机可以执行的主要动作。这一方法由客户端代码调用。

现在再来看看"我们想要什么"部分。我们决定用更多功能来丰富应用，并且幸运地在两个与我们应用无关的代码库中发现两个有意思的类，Synthesizer和Human。在Synthesizer类中，主要动作由play()方法执行。在Human类中，主要动作由speak()方法执行。为表明这两个类是外部的，将它们放在一个单独的模块中，如下所示。

```
class Synthesizer:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return 'the {} synthesizer'.format(self.name)

    def play(self):
        return 'is playing an electronic song'
```

```
class Human:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return '{} the human'.format(self.name)

    def speak(self):
        return 'says hello'
```


看起来不错，但是有一个问题：客户端仅知道如何调用execute()方法，并不知道play()和speak()。在不改变Synthesizer和Human类的前提下，我们该如何做才能让代码有效？适配器是救星！我们创建一个通用的Adapter类，将一些带不同接口的对象适配到一个统一接口中。__init__()方法的obj参数是我们想要适配的对象，adapted_methods是一个字典，键值对中的键是客户端要调用的方法，值是应该被调用的方法。

```
class Adapter:
    def __init__(self, obj, adapted_methods):
        self.obj = obj
        self.__dict__.update(adapted_methods)

    def __str__(self):
        return str(self.obj)
```

下面看看使用适配器模式的方法。列表objects容纳着所有对象。属于Computer类的可兼容对象不需要适配。可以直接将它们添加到列表中。不兼容的对象则不能直接添加。使用Adapter类来适配它们。结果是，对于所有对象，客户端代码都可以始终调用已知的execute()方法，而无需关心被使用的类之间的任何接口差别。

```
def main():
    objects = [Computer('Asus')]
    synth = Synthesizer('moog')
    objects.append(Adapter(synth, dict(execute=synth.play)))
    human = Human('Bob')
    objects.append(Adapter(human, dict(execute=human.speak)))

    for i in objects:
        print('{} {}'.format(str(i), i.execute()))
```

现在来看看适配器模式例子的完整代码（文件external.py和adapter.py），如下所示。

```
class Synthesizer:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return 'the {} synthesizer'.format(self.name)

    def play(self):
```

```
        return 'is playing an electronic song'

class Human:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return '{} the human'.format(self.name)

    def speak(self):
        return 'says hello'

from external import Synthesizer, Human

class Computer:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return 'the {} computer'.format(self.name)

    def execute(self):
        return 'executes a program'

class Adapter:
    def __init__(self, obj, adapted_methods):
        self.obj = obj
        self.__dict__.update(adapted_methods)

    def __str__(self):
        return str(self.obj)

def main():
    objects = [Computer('Asus')]
    synth = Synthesizer('moog')
    objects.append(Adapter(synth, dict(execute=synth.play)))
    human = Human('Bob')
    objects.append(Adapter(human, dict(execute=human.speak)))

    for i in objects:
        print('{} {}'.format(str(i), i.execute()))

if __name__ == "__main__":
    main()
```

执行这个例子，输出如下：

```
>>> python3 adapter.py
the Asus computer executes a program
the moog synthesizer is playing an electronic song
Bob the human says hello
```

我们设法使得`Human`和`Synthesizer`类与客户端所期望的接口兼容，且无需改变它们的源代

码。这太棒了！

这里有一个为你准备的挑战性练习，当前的实现有一个问题，当所有类都有一个属性name时，以下代码会运行失败。

```
for i in objects:
        print(i.name)
```

首先想想这段代码为什么会失败？虽然从编码的角度来看这是有意义的，但对于客户端代码来说毫无意义，客户端不应该关心“适配了什么”和“什么没有被适配”这类细节。我们只是想提供一个统一的接口。该如何做才能让这段代码生效？

> 思考一下如何将未适配部分委托给包含在适配器类中的对象。

4.5　小结

本章介绍了适配器设计模式。我们使用适配器模式让两个（或多个）不兼容接口兼容。为了引起读者的兴趣，本章先提到一个应该支持多种货币的电子商务系统[①]。我们每天都在为设备的互连、充电等使用适配器。

适配器让一件产品在制造出来之后需要应对新需求之时还能工作。Python框架Grok和第三方包Traits各自都使用了适配器模式来获得API一致性和接口兼容性。开放/封闭原则与这些方面密切相关。

在4.4节，我们看到了如何使用适配器模式，无需修改不兼容模型的源代码就能获得接口的一致性。这是通过让一个通用的适配器类完成相关工作而实现的。虽然在Python中我们可以沿袭传统方式使用子类（继承）来实现适配器模式，但这种技术是一种很棒的替代方案。

第5章中，我们将学习如何使用修饰器模式来扩展对象的行为，而无需使用子类。

① 作者的意思应该是，这是一个实际的程序员熟悉的例子，所以更能引起兴趣。——译者注

第 5 章

修饰器模式

无论何时我们想对一个对象添加额外的功能，都有下面这些不同的可选方法。

- 如果合理，可以直接将功能添加到对象所属的类（例如，添加一个新的方法）
- 使用组合
- 使用继承

与继承相比，通常应该优先选择组合，因为继承使得代码更难复用，继承关系是静态的，并且应用于整个类以及这个类的所有实例（请参考［GOF95，第31页］和网页［t.cn/RqrC8Yo］）。

设计模式为我们提供第四种可选方法，以支持动态地（运行时）扩展一个对象的功能，这种方法就是修饰器。修饰器（Decorator）模式能够以透明的方式（不会影响其他对象）动态地将功能添加到一个对象中（请参考［GOF95，第196页］）。

在许多编程语言中，使用子类化（继承）来实现修饰器模式（请参考［GOF95，第198页］）。在Python中，我们可以（并且应该）使用内置的修饰器特性。一个Python修饰器就是对Python语法的一个特定改变，用于扩展一个类、方法或函数的行为，而无需使用继承。从实现的角度来说，Python修饰器是一个可调用对象（函数、方法、类），接受一个函数对象`fin`作为输入，并返回另一个函数对象`fout`（请参考网页https://pythonconquerstheuniverse.wordpress.com/2012/04/29/python-decorators/）。这意味着可以将任何具有这些属性的可调用对象当作一个修饰器。在第1章和第2章中已经看到如何使用内置的`property`修饰器让一个方法表现为一个变量。在5.4节，我们将学习如何实现及使用我们自己的修饰器。

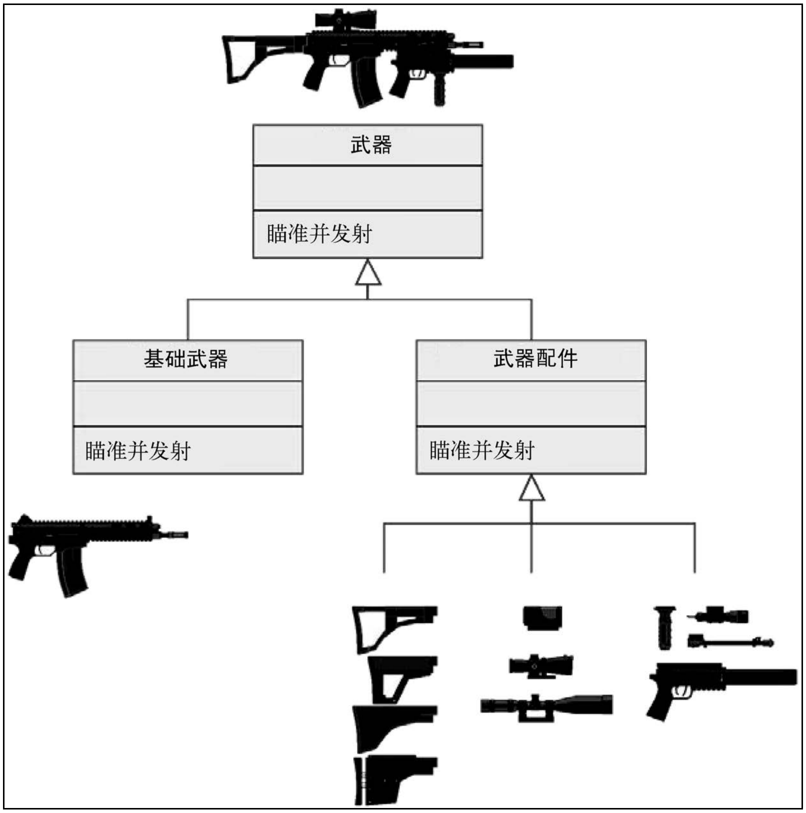
修饰器模式和Python修饰器之间并不是一对一的等价关系。Python修饰器能做的实际上比修饰器模式多得多，其中之一就是实现修饰器模式（请参考［Eckel08，第59页］和网页［t.cn/RqrlLcQ］）。

5.1 现实生活的例子

该模式虽名为修饰器，但这并不意味着它应该只用于让产品看起来更漂亮。修饰器模式通常

用于扩展一个对象的功能。这类扩展的实际例子有，给枪加一个消音器、使用不同的照相机镜头（在可拆卸镜头的照相机上）等。

下图由sourcemaking.com提供，展示了我们可以如何使用一些专用配件来修饰一把枪，使其无声、更准以及更具破坏力（请参考网页［t.cn/RqrC8Yo］）。注意，图中使用了子类化，但是在Python中，这并不是必需的，因为可以使用语言内置的修饰器特性。



5.2 软件的例子

Django框架大量地使用修饰器，其中一个例子是视图修饰器。Django的视图（View）修饰器可用于以下几种用途（请参考网页［t.cn/RqrlJbA］）。

- ❑ 限制某些HTTP请求对视图的访问
- ❑ 控制特定视图上的缓存行为
- ❑ 按单个视图控制压缩
- ❑ 基于特定HTTP请求头控制缓存

Grok框架也使用修饰器来实现不同的目标，比如下面几种情况。

- 将一个函数注册为事件订阅者
- 以特定权限保护一个方法
- 实现适配器模式

5.3　应用案例

当用于实现横切关注点（cross-cutting concerns）时，修饰器模式会大显神威（请参考［Lott14，第223页］和网页［t.cn/Rqrl6O0］）。以下是横切关注点的一些例子。

- 数据校验
- 事务处理（这里的事务类似于数据库事务，意味着要么所有步骤都成功完成，要么事务失败）
- 缓存
- 日志
- 监控
- 调试
- 业务规则
- 压缩
- 加密

一般来说，应用中有些部件是通用的，可应用于其他部件，这样的部件被看作横切关注点。

使用修饰器模式的另一个常见例子是图形用户界面（Graphical User Interface，GUI）工具集。在一个GUI工具集中，我们希望能够将一些特性，比如边框、阴影、颜色以及滚屏，添加到单个组件/部件。

5.4　实现

Python修饰器通用并且非常强大。你可以在Python官网python.org的修饰器代码库页面（请参考网页［t.cn/zRHPIq4］）中找到许多修饰器的使用样例。本节中，我们将学习如何实现一个memoization修饰器（请参考网页［t.cn/zQi9AET］）。所有递归函数都能因memoization而提速，那么来试试常用的斐波那契数列例子。使用递归算法实现斐波那契数列，直接了当，但性能问题较大，即使对于很小的数值也是如此。首先来看看朴素的实现方法（文件fibonacci_naive.py）。

```
def fibonacci(n):
    assert(n >= 0), 'n must be >= 0'
    return n if n in (0, 1) else fibonacci(n-1) + fibonacci(n-2)

if __name__ == '__main__':
```

```
from timeit import Timer
t = Timer('fibonacci(8)', 'from __main__ import fibonacci')
print(t.timeit())
```

执行一下这个例子就知道这种实现的速度有多慢了。计算第8个斐波那契数要花费17秒。运行的样例输出如下所示。

```
>>> python3 fibonacci_naive.py
16.669270177000726
```

使用memoization方法看看能否改善。在下面的代码中，我们使用一个`dict`来缓存斐波那契数列中已经计算好的数值，同时也修改传给`fabonacci()`函数的参数，计算第100个斐波那契数，而不是第8个。


```
known = {0:0, 1:1}

def fibonacci(n):
    assert(n >= 0), 'n must be >= 0'
    if n in known:
        return known[n]
    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res

if __name__ == '__main__':
    from timeit import Timer
    t = Timer('fibonacci(100)', 'from __main__ import fibonacci')
    print(t.timeit())
```

执行基于memoization的代码实现，可以看到性能得到了极大的提升，甚至对于计算大的数值性能也是可接受的。运行的样例输出如下所示。

```
>>> python3 fibonacci.py
0.31532211999729043
```

但这种方法有一些问题。虽然性能不再是一个问题，但代码也没有不使用memoization时那样简洁。如果我们决定扩展代码，加入更多的数学函数，并将其转变成一个模块，那又会是什么样的呢？假设决定加入的下一个函数是`nsum()`，该函数返回前`n`个数字的和。注意这个函数已存在于`math`模块中，名为`fsum()`，但我们也能很容易就能想到标准库中还没有、但是对我们模块有用的其他函数（例如，帕斯卡三角形、埃拉托斯特尼筛法等）。所以暂且不必在意示例函数是否已存在。使用memoization实现`nsum()`函数的代码如下所示。

```
known_sum = {0:0}

def nsum(n):
    assert(n >= 0), 'n must be >= 0'
    if n in known_sum:
        return known_sum[n]
    res = n + nsum(n-1)
    known_sum[n] = res
    return res
```

你有没有注意到其中的问题？多了一个名为known_sum的新字典，为nsum提供缓存作用，并且函数本身也比不使用memoization时的更复杂。这个模块逐步变得不必要地复杂。保持递归函数与朴素版本的一样简单，但在性能上又能与使用memoization的函数相近，这可能吗？幸运的是，确实可能，解决方案就是使用修饰器模式。

首先创建一个如下面的例子所示的memoize()函数。这个修饰器接受一个需要使用memoization的函数fn作为输入，使用一个名为known的dict作为缓存。函数functools.wraps()是一个为创建修饰器提供便利的函数；虽不强制，但推荐使用，因为它能保留被修饰函数的文档[①]和签名（请参考网页［t.cn/Rqrl0K5］)。这种情况要求参数列表*args，因为被修饰的函数可能有输入参数。如果fibonacci()和nsum()不需要任何参数，那么使用*args确实是多余的，但它们是需要参数n的。

```
import functools

def memoize(fn):
    known = dict()

    @functools.wraps(fn)
    def memoizer(*args):
        if args not in known:
            known[args] = fn(*args)
        return known[args]

    return memoizer
```

现在，对朴素版本的函数应用memoize()修饰器。这样既能保持代码的可读性又不影响性能。我们通过修饰（或修饰行）来应用一个修饰器。修饰使用@name语法，其中name是指我们想要使用的修饰器的名称。这其实只不过是一个简化修饰器使用的语法糖。我们甚至可以绕过这个语法手动执行修饰器，留给你作为练习吧。来看看下面的例子中如何对我们的递归函数使用memoize()修饰器。

```
@memoize
def nsum(n):
    '''返回前n个数字的和'''
    assert(n >= 0), 'n must be <= 0'
    return 0 if n == 0 else n + nsum(n-1)

@memoize
def fibonacci(n):
    '''返回斐波那契数列的第n个数'''
    assert(n >= 0), 'n must be >= 0'
    return n if n in (0, 1) else fibonacci(n-1) + fibonacci(n-2)
```

代码的最后一部分展示如何使用被修饰的函数，并测量其性能。measure是一个字典列表，

① 这里是指文档字符串。——译者注

用于避免代码重复。注意__name__和__doc__分别是如何展示正确的函数名称和文档字符串值的。尝试从memoize()中删除@functools.wraps(fn)修饰，看看是否仍旧如此。

```
if __name__ == '__main__':
    from timeit import Timer
    measure = [ {'exec':'fibonacci(100)', 'import':'fibonacci',
    'func':fibonacci},{'exec':'nsum(200)', 'import':'nsum',
    'func':nsum} ]
    for m in measure:
        t = Timer('{}'.format(m['exec']), 'from __main__ import
        {}'.format(m['import']))
        print('name: {}, doc: {}, executing: {}, time:
        {}'.format(m['func'].__name__, m['func'].__doc__,
        m['exec'], t.timeit()))
```

5

看看我们数学模块的完整代码（文件mymath.py）和执行时的样例输出。

```
import functools

def memoize(fn):
    known = dict()

    @functools.wraps(fn)
    def memoizer(*args):
        if args not in known:
            known[args] = fn(*args)
        return known[args]

    return memoizer

@memoize
def nsum(n):
    '''返回前n个数字的和'''
    assert(n >= 0), 'n must be >= 0'
    return 0 if n == 0 else n + nsum(n-1)

@memoize
def fibonacci(n):
    '''返回斐波那契数列的第n个数'''
    assert(n >= 0), 'n must be >= 0'
    return n if n in (0, 1) else fibonacci(n-1) + fibonacci(n-2)

if __name__ == '__main__':
    from timeit import Timer
    measure = [ {'exec':'fibonacci(100)', 'import':'fibonacci',
    'func':fibonacci},{'exec':'nsum(200)', 'import':'nsum',
    'func':nsum} ]
    for m in measure:
        t = Timer('{}'.format(m['exec']), 'from __main__ import
        {}'.format(m['import']))
        print('name: {}, doc: {}, executing: {}, time:
        {}'.format(m['func'].__name__, m['func'].__doc__,
        m['exec'], t.timeit()))
```

注意，实际的执行时间可能会有所不同。

```
>>> python3 mymath.py
name: fibonacci, doc: Returns the nth number of the Fibonacci
sequence, executing: fibonacci(100), time: 0.4169441329995607
name: nsum, doc: Returns the sum of the first n numbers,
executing: nsum(200), time: 0.4160157349997462
```

不错！这一方案同时具备可读的代码和可接受的性能。此时，你可能想争论说这不是修饰器模式，因为我们并不是在运行时应用它。被修饰的函数确实无法取消修饰，但仍然可以在运行时决定是否执行修饰器。这个有趣的练习就留给你来完成吧。

> 使用修饰器进行一层额外的封装，基于某个条件来决定是否执行真正的修饰器。

修饰器的另一个有趣的特性是可以使用多个修饰器来修饰一个函数。本章没有涉及这一特性，因此这是另一个练习，创建一个修饰器来帮助你调试递归函数，并将其应用于`nsum()`和`fibonacci()`。多个修饰器会以什么次序执行？

如果你仍未充分理解修饰器，那么我有最后一个练习留给你。修饰器`memoize()`无法修饰接受多个参数的函数。我们如何可以验证这一点？验证之后，尝试找到一种方法解决这个问题[1]。

5.5　小结

本章介绍了修饰器模式及其与Python编程语言的关联。我们使用修饰器模式来扩展一个对象的行为，无需使用继承，非常方便。Python进一步扩展了修饰器的概念，允许我们无需使用继承或组合就能扩展任意可调用对象（函数、方法或类）的行为。我们可以使用Python内置的修饰器特性。

我们看了现实中一些被修饰对象的例子，比如枪和照相机。从软件的视角来看，Django和Grok都使用了修饰器来达到不同的目标，比如控制HTTP压缩和缓存。

修饰器模式是实现横切关注点的绝佳方案，因为横切关注点通用但不太适合使用面向对象编程范式来实现。在5.3节中我们提到很多种横切关注点。事实上，5.4节演示了一个横切关注点，memoization。我们看到修饰器如何可以帮助我们保持函数简洁，同时不牺牲性能。

本章中推荐的练习可以帮助你更好地理解修饰器，这样你就能将这一强大工具用于解决许多常见的（或许不太常见的）编程问题。第6章将介绍外观模式，一种简化复杂系统访问的方式。

① 这句话可能有误。经译者测试，`memoize()`对多参函数仍然有效。——译者注

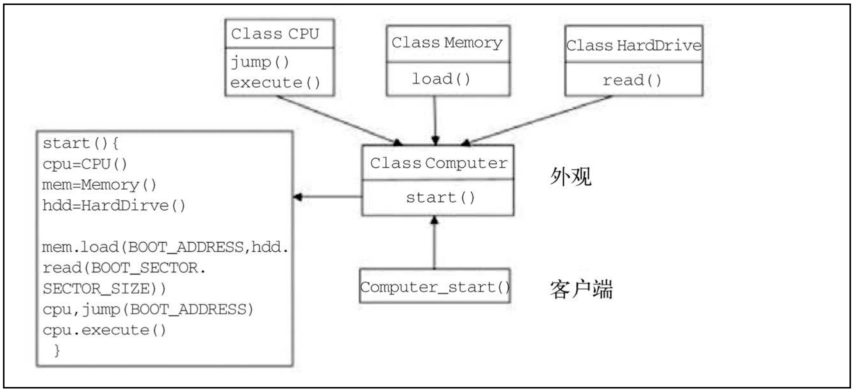
第 6 章

外观模式

6

系统会随着演化变得非常复杂，最终形成大量的（并且有时是令人迷惑的）类和交互，这种情况并不少见。许多情况下，我们并不想把这种复杂性暴露给客户端。外观设计模式有助于隐藏系统的内部复杂性，并通过一个简化的接口向客户端暴露必要的部分（请参考［Eckel08，第209页］)。本质上，外观（Facade）是在已有复杂系统之上实现的一个抽象层。

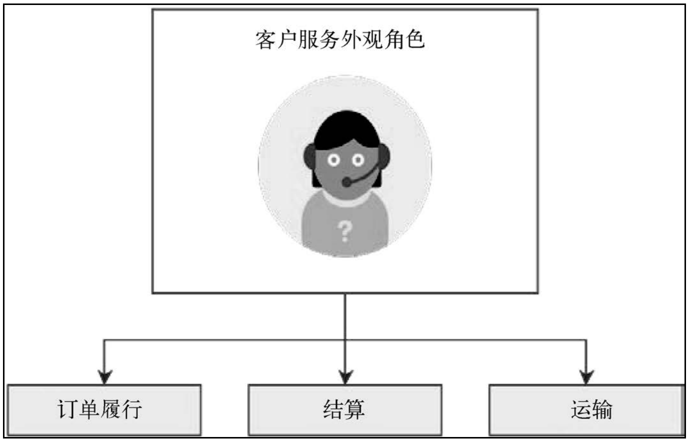
下图演示了外观的角色。这张图是Wikipedia上外观模式Java语言示例的类图表示（请参考网页［t.cn/Rqrl38m］)。计算机是一个复杂的机器，全功能运行依赖多个部件。为简化表述，这里所说的计算机是指IBM衍生的那一类，使用冯·诺依曼架构。启动一台计算机是一个相当复杂的过程。CPU、内存以及硬盘都需要加电运行；引导加载程序需要从硬盘加载到内存，CPU则必须启动操作系统内核，等等。我们不会把这些复杂性暴露给客户端，而是创造一个外观来封装整个过程，并保证所有步骤按照正确的次序运行。



从图中展示的类可知，仅`Computer`类需要暴露给客户端代码。客户端仅执行`Computer`的`start()`方法。所有其他复杂部件都由外观类`Computer`来维护。

6.1　现实生活的例子

在现实中，外观模式相当常见。当你致电一个银行或公司，通常是先被连线到客服部门，客服职员在你和业务部门（结算、技术支持、一般援助等）及帮你解决具体问题的职员之间充当一个外观的角色。下图由sourcemaking.com提供，以图表形式展示了这个例子（请参考网页［t.cn/RqrlrtI］）。



也可以将汽车或摩托车的启动钥匙视为一个外观。外观是激活一个系统的便捷方式，系统的内部则非常复杂。当然，对于其他可以通过一个简单按钮就能激活的复杂电子设备，同样可以如此看待，比如计算机。

6.2　软件的例子

django-oscar-datacash模块是Django的一个第三方组件，用于集成DataCash支付网关。该组件有一个`Gateway`类，提供对多种DataCash API的细粒度访问。在那之上，它也包含一个`Facade`类，提供粗粒度API（提供给那些不需要处理细节的人），并针对审计目的提供保存事务的能力（请参考网页［t.cn/RqrlgCG］）。

Caliendo是一个用于模拟Python API的的接口，它包含一个`facade`模块。该模块使用外观模式来完成许多不同但有用的事情（比如缓存方法），并基于传给顶层`Facade`方法的输入对象决定返回什么方法（请参考网页［t.cn/RqrlkiU］）。

6.3　应用案例

使用外观模式的最常见理由是为一个复杂系统提供单个简单的入口点。引入外观之后，客户

端代码通过简单地调用一个方法/函数就能使用一个系统。同时，内部系统并不会丢失任何功能，外观只是封装了内部系统。

不把系统的内部功能暴露给客户端代码有一个额外的好处：我们可以改变系统内部，但客户端代码不用关心这个改变，也不会受这个改变的影响。客户端代码不需要进行任何改变（请参考［Zlobin13，第44页］）。

如果你的系统包含多层，外观模式也能派上用场。你可以为每一层引入一个外观入口点，并让所有层级通过它们的外观相互通信。这提高了层级之间的松耦合性，尽可能保持层级独立（请参考［GOF95，第209页］）。

6.4 实现

假设我们想使用多服务进程方式实现一个操作系统，类似于MINIX 3（请参考网页［t.cn/h5mI2X］）或GNU Hurd（请参考网页［t.cn/RqrjZA1］）那样。多服务进程的操作系统有一个极小的内核，称为*微内核*（microkernel），它在特权模式下运行。系统的所有其他服务都遵从一种服务架构（驱动程序服务器、进程服务器、文件服务器等）。每个服务进程属于一个不同的内存地址空间，以用户模式在微内核之上运行。这种方式的优势是操作系统更能容错、更加可靠、更加安全。例如，由于所有驱动程序都以用户模式在一个驱动服务进程之上运行，所以某个驱动程序中的一个bug并不能让整个系统崩溃，也无法影响到其他服务进程。其劣势则是性能开销和系统编程的复杂性，因为服务进程和微内核之间，还有独立的服务进程之间，使用消息传递方式进行通信。消息传递比宏内核（如Linux）所使用的共享内存模型更加复杂（请参考网页［t.cn/RqrjAK8］）。

我们从`Server`接口[①]开始实现，使用一个`Enum`类型变量来描述一个服务进程的不同状态，使用`abc`模块来禁止对`Server`接口直接进行初始化，并强制子类实现关键的`boot()`和`kill()`方法。这里假设每个服务进程的启动、关闭及重启都相应地需要不同的动作。如果你以前没用过`abc`模块，请记住以下几个重要事项。

- 我们需要使用`metaclass`关键字来继承`ABCMeta`。
- 使用`@abstractmethod`修饰器来声明`Server`的所有子类都应（强制性地）实现哪些方法。

尝试移除一个子类的`boot()`或`kill()`方法，看看会发生什么。移除`@abstractmethod`修饰器之后再试试。一切如你所料吗？

我们来思考以下这段代码。

① 这里的“接口”并非指语法上的interface，而是指一个不能直接实例化的类。——译者注

```
State = Enum('State', 'new running sleeping restart zombie')

class Server(metaclass=ABCMeta):
    @abstractmethod
    def __init__(self):
        pass

    def __str__(self):
        return self.name

    @abstractmethod
    def boot(self):
        pass

    @abstractmethod
    def kill(self, restart=True):
        pass
```

一个模块化的操作系统可以有很多有意思的服务进程，包括文件服务进程、进程服务进程、身份验证服务进程、网络服务进程和图形/窗口服务进程等。下面这个例子包含两个存根服务进程（`FileServer`和`ProcessServer`）。除了`Server`接口要求实现的方法之外，每个服务进程还可以具有自己特有的方法。例如，`FileServer`有一个`create_file()`方法用于创建文件，`ProcessServer`有一个`create_process()`方法用于创建进程。

```
class FileServer(Server):
    def __init__(self):
        '''初始化文件服务进程要求的操作'''
        self.name = 'FileServer'
        self.state = State.new

    def boot(self):
        print('booting the {}'.format(self))
        '''启动文件服务进程要求的操作'''
        self.state = State.running

def kill(self, restart=True):
        print('Killing {}'.format(self))
        '''终止文件服务进程要求的操作'''
        self.state = State.restart if restart else State.zombie

    def create_file(self, user, name, permissions):
        '''检查访问权限的有效性和用户权限等'''

        print("trying to create the file '{}' for user '{}' with permissions
        {}".format(name, user, permissions))

class ProcessServer(server):
    def __init__(self):
        '''初始化进程服务进程要求的操作'''
        self.name = 'ProcessServer'
        self.state = State.new
```

```
    def boot(self):
        print('booting the {}'.format(self))
        '''启动进程服务进程要求的操作'''
        self.state = State.running

    def kill(self, restart=True):
        print('Killing {}'.format(self))
        '''终止进程服务进程要求的操作'''
        self.state = State.restart if restart else State.zombie

    def create_process(self, user, name):
        '''检查用户权限和生成PID等'''

        print("trying to create the process '{}' for user '{}'".format(name, user))
```

OperatingSystem类是一个外观。__init__()中创建所有需要的服务进程实例。start()方法是系统的入口点，供客户端代码使用。如果需要，可以添加更多的包装方法作为服务的访问点，比如包装方法create_file()和create_process()。从客户端的角度来看，所有服务都是由OperatingSystem类提供的。客户端并不应该被不必要的细节所干扰，比如，服务进程的存在和每个服务进程的责任。

6

```
class OperatingSystem:
    '''外观'''
    def __init__(self):
        self.fs = FileServer()
        self.ps = ProcessServer()

    def start(self):
        [i.boot() for i in (self.fs, self.ps)]

    def create_file(self, user, name, permissions):
        return self.fs.create_file(user, name, permissions)

    def create_process(self, user, name):
        return self.ps.create_process(user, name)
```

在下面的完整代码清单中（文件facade.py），可以看到许多模拟的类和服务进程，它们的存在是为了让读者了解系统运转要求哪些抽象（User、Process和File等）和服务进程（WindowServer和NetworkServer等）。推荐至少实现系统的一个服务来练习一下（例如，文件创建）。可随意改变接口和方法签名来满足你的需求，但要确保在改变之后，客户端代码不需要知道OperatingSystem外观类之外的任何对象。

```
from enum import Enum
from abc import ABCMeta, abstractmethod

State = Enum('State', 'new running sleeping restart zombie')

class User:
    pass

class Process:
```

```
    pass

class File:
    pass

class Server(metaclass=ABCMeta):
    @abstractmethod
    def __init__(self):
        pass

    def __str__(self):
        return self.name

    @abstractmethod
    def boot(self):
        pass

    @abstractmethod
    def kill(self, restart=True):
        pass

class FileServer(Server):
    def __init__(self):
        '''初始化文件服务进程要求的操作'''
        self.name = 'FileServer'
        self.state = State.new

    def boot(self):
        print('booting the {}'.format(self))
        '''启动文件服务进程要求的操作'''
        self.state = State.running

    def kill(self, restart=True):
        print('Killing {}'.format(self))
        '''终止文件服务进程要求的操作'''
        self.state = State.restart if restart else State.zombie

    def create_file(self, user, name, permissions):
        '''检查访问权限的有效性、用户权限等'''

        print("trying to create the file '{}' for user '{}' with permissions
        {}".format(name, user, permissions))

class ProcessServer(Server):
    def __init__(self):
        '''初始化进程服务进程要求的操作'''
        self.name = 'ProcessServer'
        self.state = State.new

    def boot(self):
        print('booting the {}'.format(self))
        '''启动进程服务进程要求的操作'''
        self.state = State.running
```

```
    def kill(self, restart=True):
        print('Killing {}'.format(self))
        '''终止进程服务进程要求的操作'''
        self.state = State.restart if restart else State.zombie

    def create_process(self, user, name):
        '''检查用户权限和生成PID等'''

        print("trying to create the process '{}' for user '{}'".format(name, user))

class WindowServer:
    pass

class NetworkServer:
    pass

class OperatingSystem:
    '''外观'''
    def __init__(self):
        self.fs = FileServer()
        self.ps = ProcessServer()

    def start(self):
        [i.boot() for i in (self.fs, self.ps)]

    def create_file(self, user, name, permissions):
        return self.fs.create_file(user, name, permissions)

    def create_process(self, user, name):
        return self.ps.create_process(user, name)

def main():
    os = OperatingSystem()
    os.start()
    os.create_file('foo', 'hello', '-rw-r-r')
    os.create_process('bar', 'ls /tmp')

if __name__ == '__main__':
    main()
```

执行这个例子会显示两个存根服务进程的启动信息。

```
>>> python3 facade.py
booting the FileServer
booting the ProcessServer
trying to create the file 'hello' for user 'foo' with permissions - rw-r-r
trying to create the process 'ls /tmp' for user 'bar'
```

外观类`OperatingSystem`起到了很好的作用。客户端代码可以创建文件和进程，而无需知道操作系统的内部细节，比如，多个服务进程的存在。准确点说是客户端可以调用方法来创建文件和进程，但是目前它们是模拟的。如果感兴趣，你可以实现这两个方法之一作为练习，或者两个都实现。

6.5　小结

本章中，我们学习了如何使用外观模式。在客户端代码想要使用一个复杂系统但又不关心系统复杂性之时，这种模式是为复杂系统提供一个简单接口的理想方式。一台计算机是一个外观，因为当我们使用它时需要做的事情仅是按一个按钮来启动它；其余的所有硬件复杂性都用户无感知地交由BIOS、引导加载程序以及其他系统软件来处理。现实生活中外观的例子更多，比如，我们所致电的银行或公司客服部门，还有启动机动车所使用的钥匙。

我们讨论了两个使用外观的Django第三方组件：`django-oscar-datacash`和`Caliendo`。前者使用外观模式来提供一个简单的DataCash API以及保存事务的能力，后者为多种目的使用了外观，比如，缓存、基于输入对象的类型决定应该返回什么。

我们讲解了外观基本的应用案例，并以多服务进程操作系统使用的接口实现来结束本章内容。外观是一种隐藏系统复杂性的优雅方式，因为多数情况下客户端代码并不应该关心系统的这些细节。

第7章中，我们将学习如何使用享元设计模式来复用对象，提高系统的资源利用率。

第 7 章

享元模式

由于对象创建的开销，面向对象的系统可能会面临性能问题。性能问题通常在资源受限的嵌入式系统中出现，比如智能手机和平板电脑。大型复杂系统中也可能会出现同样的问题，因为要在其中创建大量对象（也可能是用户），这些对象需要同时并存。

这个问题之所以会发生，是因为当我们创建一个新对象时，需要分配额外的内存。虽然虚拟内存理论上为我们提供了无限制的内存空间，但现实却并非如此。如果一个系统耗尽了所有的物理内存，就会开始将内存页替换到二级存储设备，通常是*硬盘驱动器*（Hard Disk Drive，HDD）。在多数情况下，由于内存和硬盘之间的性能差异，这是不能接受的。*固态硬盘*（Solid State Drive，SSD）的性能一般比硬盘更好，但并非人人都使用SSD，SSD并不会很快全面替代硬盘（请参考网页［t.cn/RqrjS0E］）。

除内存使用之外，计算性能也是一个考虑点。图形软件，包括计算机游戏，应该能够极快地渲染3D信息（例如，有成千上万棵树的森林或满是士兵的村庄）。如果一个3D地带的每个对象都是单独创建，未使用数据共享，那么性能将是无法接受的（请参考网页［t.cn/Rqrj9qa］）。

作为软件工程师，我们应该编写更好的软件来解决软件问题，而不是要求客户购买更多更好的硬件。享元设计模式通过为相似对象引入数据共享来最小化内存使用，提升性能（请参考网页［t.cn/RqrjNF3］）。一个*享元*（Flyweight）就是一个包含状态独立的不可变（又称固有的）数据的共享对象。依赖状态的可变（又称非固有的）数据不应是享元的一部分，因为每个对象的这种信息都不同，无法共享。如果享元需要非固有的数据，应该由客户端代码显式地提供（请参考［GOF95，第219页］和网页［t.cn/RqrjOX3］）。

用一个例子可能有助于解释实际应用场景中如何使用享元模式。假设我们正在设计一个性能关键的游戏，例如*第一人称射击*（First-Person Shooter，FPS）游戏。在FPS游戏中，玩家（士兵）共享一些状态，如外在表现和行为。例如，在《反恐精英》游戏中，同一团队（反恐精英或恐怖分子）的所有士兵看起来都是一样的（外在表现）。同一个游戏中，（两个团队的）所有士兵都有一些共同的动作，比如，跳起、低头等（行为）。这意味着我们可以创建一个享元来包含所有共同的数据。当然，士兵也有许多因人而异的可变数据，这些数据不是享元的一部分，比如，枪支、健康状况和地理位置等。

7.1　现实生活的例子

享元模式是一个用于优化的设计模式。因此，要找一个合适的现实生活的例子不太容易。我们可以把享元看作现实生活中的缓存区。例如，许多书店都有专用的书架来摆放最新和最流行的出版物。这就是一个缓存区，你可以先在这些专用书架上看看有没有正在找的书籍，如果没找到，则可以让图书管理员来帮你。

7.2　软件的例子

Exaile音乐播放器（请参考网页［t.cn/RqrjYHQ］）使用享元来复用通过相同URL识别的对象（在这里是指音乐歌曲）。创建一个与已有对象的URL相同的新对象是没有意义的，所以复用相同的对象来节约资源（请参考网页［http://t.cn/RqrjQWr］）。

Peppy是一个用Python语言实现的类XEmacs编辑器（请参考网页［t.cn/hbhSda］），它使用享元模式存储`major mode`状态栏的状态。这是因为除非用户修改，否则所有状态栏共享相同的属性（请参考网页［t.cn/Rqrjm9y］）。

7.3　应用案例

享元旨在优化性能和内存使用。所有嵌入式系统（手机、平板电脑、游戏终端和微控制器等）和性能关键的应用（游戏、3D图形处理和实时系统等）都能从其获益。

若想要享元模式有效，需要满足GoF的《设计模式》一书罗列的以下几个条件。

- 应用需要使用大量的对象。
- 对象太多，存储/渲染它们的代价太大。一旦移除对象中的可变状态（因为在需要之时，应该由客户端代码显式地传递给享元），多组不同的对象可被相对更少的共享对象所替代。
- 对象ID对于应用不重要。对象共享会造成ID比较的失败，所以不能依赖对象ID（那些在客户端代码看来不同的对象，最终具有相同的ID）。

7.4　实现

由于之前已提到树的例子，那么就来看看如何实现它。在这个例子中，我们将构造一小片水果树的森林，小到能确保在单个终端页面中阅读整个输出。然而，无论你构造的森林有多大，内存分配都保持相同。下面这个`Enum`类型变量描述三种不同种类的水果树。

```
TreeType = Enum('TreeType', 'apple_tree cherry_tree peach_tree')
```

在深入代码之前，我们稍稍解释一下memoization与享元模式之间的区别。memoization是一种优化技术，使用一个缓存来避免重复计算那些在更早的执行步骤中已经计算好的结果。memoization并不是只能应用于某种特定的编程方式，比如面向对象编程（Object-Oriented Programming，OOP）。在Python中，memoization可以应用于方法和简单的函数。享元则是一种特定于面向对象编程优化的设计模式，关注的是共享对象数据。

在Python中，享元可以以多种方式实现，但我发现这个例子中展示的实现非常简洁。pool变量是一个对象池（换句话说，是我们的缓存）。注意：pool是一个类属性（类的所有实例共享的一个变量，请参考网页［t.cn/zHwpgFe］）。使用特殊方法__new__（这个方法在__init__之前被调用），我们把Tree类变换成一个元类，元类支持自引用。这意味着cls引用的是Tree类（请参考［Lott14，第99页］）。当客户端要创建Tree的一个实例时，会以tree_type参数传递树的种类。树的种类用于检查是否创建过相同种类的树。如果是，则返回之前创建的对象；否则，将这个新的树种添加到池中，并返回相应的新对象，如下所示。

```
    def __new__(cls, tree_type):
        obj = cls.pool.get(tree_type, None)
        if not obj:
            obj = object.__new__(cls)
            cls.pool[tree_type] = obj
            obj.tree_type = tree_type
        return obj
```


方法render()用于在屏幕上渲染一棵树。注意，享元不知道的所有可变（外部的）信息都需要由客户端代码显式地传递。在当前案例中，每棵树都用到一个随机的年龄和一个x，y形式的位置。为了让render()更加有用，有必要确保没有树会被渲染到另一个棵之上。你可以考虑把这个作为练习。如果你想让渲染更加有趣，可以使用一个图形工具包，比如Tkinter或Pygame。

```
def render(self, age, x, y):
        print('render a tree of type {} and age {} at ({}, {})'.format(self.tree_type,
        age, x, y))
```

main()函数展示了我们可以如何使用享元模式。一棵树的年龄是1到30年之间的一个随机值。坐标使用1到100之间的随机值。虽然渲染了18棵树，但仅分配了3棵树的内存。输出的最后一行证明当使用享元时，我们不能依赖对象的ID。函数id()会返回对象的内存地址。Python规范并没有要求id()返回对象的内存地址，只是要求id()为每个对象返回一个唯一性ID，不过CPython（Python的官方实现）正好使用对象的内存地址作为对象唯一性ID。在我们的例子中，即使两个对象看起来不相同，但是如果它们属于同一个享元家族（在这里，家族由tree_type定义），那么它们实际上有相同的ID。当然，不同ID的比较仍然可用于不同家族的对象，但这仅在客户端知道实现细节的情况下才可行（通常并非如此）。

```
def main():
    rnd = random.Random()
    age_min, age_max = 1, 30    # 单位为年
```

```
    min_point, max_point = 0, 100
    tree_counter = 0

    for _ in range(10):
        t1 = Tree(TreeType.apple_tree)
        t1.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point))
        tree_counter += 1

    for _ in range(3):
        t2 = Tree(TreeType.cherry_tree)
        t2.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point))
        tree_counter += 1

    for _ in range(5):
        t3 = Tree(TreeType.peach_tree)
        t3.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point))
        tree_counter += 1

    print('trees rendered: {}'.format(tree_counter))
    print('trees actually created: {}'.format(len(Tree.pool)))

    t4 = Tree(TreeType.cherry_tree)
    t5 = Tree(TreeType.cherry_tree)
    t6 = Tree(TreeType.apple_tree)
    print('{} == {}? {}'.format(id(t4), id(t5), id(t4) == id(t5)))
    print('{} == {}? {}'.format(id(t5), id(t6), id(t5) == id(t6)))
```

下面完整的代码清单（文件flyweight.py）将给出享元模式如何实现及使用的完整描述。

```
import random
from enum import Enum

TreeType = Enum('TreeType', 'apple_tree cherry_tree peach_tree')

class Tree:
    pool = dict()

    def __new__(cls, tree_type):
        obj = cls.pool.get(tree_type, None)
        if not obj:
            obj = object.__new__(cls)
            cls.pool[tree_type] = obj
            obj.tree_type = tree_type
        return obj

    def render(self, age, x, y):
        print('render a tree of type {} and age {} at ({}, {})'.format(self.tree_type,
        age, x, y))
```

```
def main():
    rnd = random.Random()
    age_min, age_max = 1, 30    # 单位为年
    min_point, max_point = 0, 100
    tree_counter = 0

    for _ in range(10):
        t1 = Tree(TreeType.apple_tree)
        t1.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point))
        tree_counter += 1

    for _ in range(3):
        t2 = Tree(TreeType.cherry_tree)
        t2.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point))
        tree_counter += 1

    for _ in range(5):
        t3 = Tree(TreeType.peach_tree)
        t3.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point))
        tree_counter += 1

    print('trees rendered: {}'.format(tree_counter))
    print('trees actually created: {}'.format(len(Tree.pool)))

    t4 = Tree(TreeType.cherry_tree)
    t5 = Tree(TreeType.cherry_tree)
    t6 = Tree(TreeType.apple_tree)
    print('{} == {}? {}'.format(id(t4), id(t5), id(t4) == id(t5)))
    print('{} == {}? {}'.format(id(t5), id(t6), id(t5) == id(t6)))

if __name__ == '__main__':
    main()
```

执行上面的示例程序会显示被渲染对象的类型、随机年龄以及坐标，还有相同/不同家族享元对象ID的比较结果。你在执行这个程序时别指望能看到与下面相同的输出，因为年龄和坐标是随机的，对象ID也依赖内存映射。

```
>>> python3 flyweight.py
render a tree of type TreeType.apple_tree and age 4 at (88, 19)
render a tree of type TreeType.apple_tree and age 18 at (31, 35)
render a tree of type TreeType.apple_tree and age 7 at (54, 23)
render a tree of type TreeType.apple_tree and age 3 at (9, 11)
render a tree of type TreeType.apple_tree and age 2 at (93, 6)
render a tree of type TreeType.apple_tree and age 12 at (3, 49)
render a tree of type TreeType.apple_tree and age 10 at (5, 65)
render a tree of type TreeType.apple_tree and age 6 at (19, 16)
```

```
render a tree of type TreeType.apple_tree and age 2 at (21, 32)
render a tree of type TreeType.apple_tree and age 21 at (87, 79)
render a tree of type TreeType.cherry_tree and age 24 at (94, 31)
render a tree of type TreeType.cherry_tree and age 14 at (92, 37)
render a tree of type TreeType.cherry_tree and age 14 at (9, 88)
render a tree of type TreeType.peach_tree and age 23 at (44, 90)
render a tree of type TreeType.peach_tree and age 16 at (15, 59)
render a tree of type TreeType.peach_tree and age 1 at (81, 98)
render a tree of type TreeType.peach_tree and age 13 at (67, 63)
render a tree of type TreeType.peach_tree and age 12 at (69, 42)
trees rendered: 18
trees actually created: 3
140322427827480 == 140322427827480? True
140322427827480 == 140322427709088? False
```

如果你想更多地练习一下享元模式，可以尝试实现本章提到的FPS士兵。思考一下哪些数据应该是享元的一部分（不可变的、内部的），哪些数据不应该是（可变的、外部的）。

7.5　小结

本章中，我们学习了享元模式。在我们想要优化内存使用提高应用性能之时，可以使用享元。在所有内存受限（想一想嵌入式系统）或关注性能的系统（比如图形软件和电子游戏）中，这一点相当重要。基于GTK+的Exaile音乐播放器使用享元来避免对象复制，Peppy文本编辑器则使用享元来共享状态栏的属性。

一般来说，在应用需要创建大量的计算代价大但共享许多属性的对象时，可以使用享元。重点在于将不可变（可共享）的属性与可变的属性区分开。我们实现了一个树渲染器，支持三种不同的树家族。通过显式地向`render()`方法提供可变的年龄和`x`，`y`属性，我们成功地仅创建了3个不同的对象，而不是18个。虽然那看起来似乎没什么了不起，但是想象一下，如果是2000棵树而不是18棵树，那又会怎样呢？

第8章将学习一种非常流行的设计模式，用于解耦处理用户界面的代码与处理（业务）逻辑的代码，这种模式就是模型–视图–控制器模式。

第 8 章

模型–视图–控制器模式

关注点分离（Separation of Concerns，SoC）原则是软件工程相关的设计原则之一。SoC原则背后的思想是将一个应用切分成不同的部分，每个部分解决一个单独的关注点。分层设计中的层次（数据访问层、业务逻辑层和表示层等）即是关注点的例子。使用SoC原则能简化软件应用的开发和维护（请参考网页［t.cn/RqrjewK］）。

模型–视图–控制器（Model-View-Controller，MVC）模式是应用到面向对象编程的Soc原则。模式的名称来自用来切分软件应用的三个主要部分，即模型部分、视图部分和控制器部分。MVC被认为是一种架构模式而不是一种设计模式。架构模式与设计模式之间的区别在于前者比后者的范畴更广。然而，MVC太重要了，不能仅因为这个原因就跳过不说。即使我们从不需要从头实现它，也需要熟悉它，因为所有常见框架都使用了MVC或者是其略微不同的版本（之后会详述）。

模型是核心的部分，代表着应用的信息本源，包含和管理（业务）逻辑、数据、状态以及应用的规则。视图是模型的可视化表现。视图的例子有，计算机图形用户界面、计算机终端的文本输出、智能手机的应用图形界面、PDF文档、饼图和柱状图等。视图只是展示数据，并不处理数据。控制器是模型与视图之间的链接/粘附。模型与视图之间的所有通信都通过控制器进行（请参考［GOF95，第14页］、网页［t.cn/RqrjF4G］和网页［t.cn/RPrOUPr］）。

对于将初始屏幕渲染给用户之后使用MVC的应用，其典型使用方式如下所示。

- 用户通过单击（键入、触摸等）某个按钮触发一个视图
- 视图把用户操作告知控制器
- 控制器处理用户输入，并与模型交互
- 模型执行所有必要的校验和状态改变，并通知控制器应该做什么
- 控制器按照模型给出的指令，指导视图适当地更新和显示输出

你可能想知道为什么控制器部分是必要的？我们能跳过它吗？能，但那样我们将失去MVC提供的一大优势：无需修改模型就能使用多个视图的能力（甚至可以根据需要同时使用多个视图）。为了实现模型与其表现之间的解耦，每个视图通常都需要属于它的控制器。如果模型直接与特定视图通信，我们将无法对同一个模型使用多个视图（或者至少无法以简洁模块化的方式实现）。

8.1　现实生活的例子

MVC是应用于面向对象编程的SoC原则。SoC原则在现实生活中的应用有很多。例如，如果你造一栋新房子，通常会请不同的专业人员来完成以下工作。

- 安装管道和电路
- 粉刷房子

另一个例子是餐馆。在一个餐馆中，服务员接收点菜单并为顾客上菜，但是饭菜由厨师烹饪（请参考网页［t.cn/RqrYh1I］）。

8.2　软件的例子

Web框架web2py（请参考网页［t.cn/RqrYZwy］）是一个支持MVC模式的轻量级Python框架。若你还未尝试过web2py，我推荐你试用一下，安装过程极其简单，你要做的就是下载安装包并执行一个Python文件（web2py.py）。在该项目的网页上有很多例子演示了在web2py中如何使用MVC（请参考网页［t.cn/RqrYADU］）。

Django也是一个MVC框架，但是它使用了不同的命名约定。在此约定下，控制器被称为视图，视图被称为模板。Django使用名称模型–模板–视图（Model-Template-View，MTV）来替代MVC。依据Django的设计者所言，视图是描述哪些数据对用户可见。因此，Django把对应一个特定URL的Python回调函数称为视图。Django中的“模板”用于把内容与其展现分开，其描述的是用户看到数据的方式，而不是哪些数据可见（请参考网页［t.cn/RwRJZ87］）。

8.3　应用案例

MVC是一个非常通用且大有用处的设计模式。实际上，所有流行的Web框架（Django、Rails和Yii）和应用框架（iPhone SDK、Android和QT）都使用了MVC或者其变种，其变种包括模式–视图–适配器（Model-View-Adapter，MVA）、模型–视图–演示者（Model-View-Presenter，MVP）等。然而，即使我们不使用这些框架，凭自己实现这一模式也是有意义的，因为这一模式提供了以下这些好处。

- 视图与模型的分离允许美工一心搞UI部分，程序员一心搞开发，不会相互干扰。
- 由于视图与模型之间的松耦合，每个部分可以单独修改/扩展，不会相互影响。例如，添加一个新视图的成本很小，只要为其实现一个控制器就可以了。
- 因为职责明晰，维护每个部分也更简单。

在从头开始实现MVC时，请确保创建的模型很智能，控制器很瘦，视图很傻瓜（请参考

［Zlobin13，第9页］)。

可以将具有以下功能的模型视为智能模型。

- 包含所有的校验/业务规则/逻辑
- 处理应用的状态
- 访问应用数据（数据库、云或其他）
- 不依赖UI

可以将符合以下条件的控制器视为瘦控制器。

- 在用户与视图交互时，更新模型
- 在模型改变时，更新视图
- 如果需要，在数据传递给模型/视图之前进行处理
- 不展示数据
- 不直接访问应用数据
- 不包含校验/业务规则/逻辑

可以将符合以下条件的视图视为傻瓜视图。

- 展示数据
- 允许用户与其交互
- 仅做最小的数据处理，通常由一种模板语言提供处理能力（例如，使用简单的变量和循环控制）
- 不存储任何数据
- 不直接访问应用数据
- 不包含校验/业务规则/逻辑

如果你正在从头实现MVC，并且想弄清自己做得对不对，可以尝试回答以下两个关键问题。

- 如果你的应用有GUI，那它可以换肤吗？易于改变它的皮肤/外观以及给人的感受吗？可以为用户提供运行期间改变应用皮肤的能力吗？如果这做起来并不简单，那就意味着你的MVC实现在某些地方存在问题（请参考网页［t.cn/RqrjF4G］)。
- 如果你的应用没有GUI（例如，是一个终端应用），为其添加GUI支持有多难？或者，如果添加GUI没什么用，那么是否易于添加视图从而以图表（饼图、柱状图等）或文档（PDF、电子表格等）形式展示结果？如果因此而作出的变更不小（小的变更是，在不变更模型的情况下，创建控制器并绑定到视图），那你的MVC实现就有些不对了。

如果你确信这两个条件都已满足，那么与未使用MVC模式的应用相比，你的应用会更灵活、更好维护。

8.4　实现

我可以使用任意常见框架来演示如何使用MVC，但觉得那样的话，读者对MVC的理解会不完整。因此我决定使用一个非常简单的示例来展示如何从头实现MVC，这个示例是名人名言打印机。想法极其简单：用户输入一个数字，然后就能看到与这个数字相关的名人名言。名人名言存储在一个quotes元组中。这种数据通常是存储在数据库、文件或其他地方，只有模型能够直接访问它。

我们从下面的代码开始考虑这个例子。

```
quotes = ('A man is not complete until he is married. Then he is finished.',
          'As I said before, I never repeat myself.',
          'Behind a successful man is an exhausted woman.',
          'Black holes really suck...', 'Facts are stubborn things.')
```

模型极为简约，只有一个get_quote()方法，基于索引n从quotes元组中返回对应的名人名言（字符串）。注意，n可以小于等于0，因为这种索引方式在Python中是有效的。本节末尾准备了练习，供你改进这个方法的行为。

```
class QuoteModel:
    def get_quote(self, n):
        try:
            value = quotes[n]
        except IndexError as err:
            value = 'Not found!'
        return value
```

视图有三个方法，分别是show()、error()和select_quote()。show()用于在屏幕上输出一句名人名言（或者输出提示信息Not found!）；error()用于在屏幕上输出一条错误消息；select_quote()用于读取用户的选择，如以下代码所示。

```
class QuoteTerminalView:
    def show(self, quote):
        print('And the quote is: "{}"'.format(quote))

    def error(self, msg):
        print('Error: {}'.format(msg))

    def select_quote(self):
        return input('Which quote number would you like to see? ')
```

控制器负责协调。__init__()方法初始化模型和视图。run()方法校验用户提供的名言索引，然后从模型中获取名言，并返回给视图展示，如以下代码所示。

```
class QuoteTerminalController:
    def __init__(self):
        self.model = QuoteModel()
        self.view = QuoteTerminalView()
```

```
    def run(self):
        valid_input = False
        while not valid_input:
            n = self.view.select_quote()
            try:
                n = int(n)
            except ValueError as err:
                self.view.error("Incorrect index '{}'".format(n))
            else:
                valid_input = True
        quote = self.model.get_quote(n)
        self.view.show(quote)
```

最后，但同样重要的是，`main()`函数初始化并触发控制器，如以下代码所示。

```
def main():
    controller = QuoteTerminalController()
    while True:
        controller.run()
```

以下是该示例的完整代码（文件mvc.py）。

```
quotes = ('A man is not complete until he is married. Then he is finished.',
        'As I said before, I never repeat myself.',
        'Behind a successful man is an exhausted woman.',
        'Black holes really suck...', 'Facts are stubborn things.')

class QuoteModel:
    def get_quote(self, n):
        try:
            value = quotes[n]
        except IndexError as err:
            value = 'Not found!'
        return value

class QuoteTerminalView:
    def show(self, quote):
        print('And the quote is: "{}"'.format(quote))

    def error(self, msg):
        print('Error: {}'.format(msg))

    def select_quote(self):
        return input('Which quote number would you like to see? ')

class QuoteTerminalController:
    def __init__(self):
        self.model = QuoteModel()
        self.view = QuoteTerminalView()

    def run(self):
        valid_input = False
        while not valid_input:
```

```
            try:
                n = self.view.select_quote()
                n = int(n)
                vaild_input = True
            except ValueError as err:
                self.view.error("Incorrect index '{}'".format(n))
        quote = self.model.get_quote(n)
        self.view.show(quote)

def main():
    controller = QuoteTerminalController()
    while True:
        controller.run()

if __name__ == '__main__':
    main()
```

下面是mvc.py的样例执行，展示了程序如何处理错误以及为用户输出名言。

```
>>> python3 mvc.py
Which quote number would you like to see? a
Error: Incorrect index 'a'
Which quote number would you like to see? 40
And the quote is: "Not found!"
Which quote number would you like to see? 0
And the quote is: "A man is not complete until he is married. Then he is
finished."
Which quote number would you like to see? 3
And the quote is: "Black holes really suck..."
```

当然，你不会（也不应该）就此止步。坚持多写代码，还有很多有意思的想法可以试验，比如以下这些。

- 仅允许用户使用大于或等于1的索引，程序会显得更加友好。为此，你也需要修改`get_quote()`。
- 使用Tkinter、Pygame或Kivy之类的GUI框架来添加一个图形化视图。程序如何模块化？可以在程序运行期间决定使用哪个视图吗？
- 让用户可以选择键入某个键（例如，`r`键）随机地看一句名言。
- 索引校验目前是在控制器中完成的。这个方式好吗？如果你编写了另一个视图，需要它自己的控制器，那又该怎么办呢？试想一下，为了让索引校验的代码被所有控制/视图复用，将索引校验移到模型中进行，需要做哪些变更？
- 对这个例子进行扩展，使其变得像一个创建、读取、更新、删除（Create, Read, Update, Delete，CURD）应用。你应该能够输入新的名言，删除已有的名言，以及修改名言。

8.5 小结

本章中，我们学习了MVC模式。MVC是一个非常重要的设计模式，用于将应用组织成三个部分：模型、视图和控制器。

每个部分都有明确的职责。模型负责访问数据，管理应用的状态。视图是模型的外在表现。视图并非必须是图形化的；文本输出也是一种好视图。控制器是模型与视图之间的连接。MVC的恰当使用能确保最终产出的应用易于维护、易于扩展。

MVC模式是应用到面向对象编程的SoC原则。这一原则类似于一栋新房子如何建造，或一个餐馆如何运营。

Python框架web2py使用MVC作为核心架构理念。即使是最简单的web2py例子也使用了MVC来实现模块化和可维护性。Django也是一个MVC框架，但它使用的名称是MTV。

使用MVC时，请确保创建智能的模型（核心功能）、瘦控制器（实现视图与模型之间通信的能力）以及傻瓜式的视图（外在表现，最小化逻辑处理）。

在8.4节中，我们学习了如何从零开始实现MVC，为用户展示有趣的名人名言。这与罗列一个RSS源的所有文章所要求的功能没什么两样，如果你对其他推荐练习不感兴趣，可以练习实现这个。

第9章将学习如何使用代理设计模式来实现一个额外的保护层，为接口提供安全性。

第 9 章

代理模式

在某些应用中，我们想要在访问某个对象之前执行一个或多个重要的操作，例如，访问敏感信息——在允许用户访问敏感信息之前，我们希望确保用户具备足够的权限。操作系统中也存在类似的情况，用户必须具有管理员权限才能在系统中安装新程序。

上面提到的重要操作不一定与安全问题相关。延迟初始化（请参考网页［t.cn/Ryf47bV］）是另一个案例：我们想要把一个计算成本较高的对象的创建过程延迟到用户首次真正使用它时才进行。

这类操作通常使用代理设计模式（Proxy design pattern）来实现。该模式因使用代理（又名替代，surrogate）对象在访问实际对象之前执行重要操作而得其名。以下是四种不同的知名代理类型（请参考［GOF95，第234页］和网页［t.cn/RqrYEn9］）。

- 远程代理：实际存在于不同地址空间（例如，某个网络服务器）的对象在本地的代理者。
- 虚拟代理：用于懒初始化，将一个大计算量对象的创建延迟到真正需要的时候进行。
- 保护/防护代理：控制对敏感对象的访问。
- 智能（引用）代理：在对象被访问时执行额外的动作。此类代理的例子包括引用计数和线程安全检查。

我发现虚拟代理非常有用，所以现在通过一个例子来看看可以如何实现它。在9.4节中将学习如何创建防护代理。

使用Python来创建虚拟代理存在很多方式，但我始终喜欢地道的/符合Python风格的实现。这里展示的代码源自网站stackoverflow.com用户Cyclone的一个超赞回答（请参考网页［t.cn/RqrYudC］）。为避免混淆，我先说明一下，在本节中，术语特性（property）、变量（variable）、属性（attribute）可相互替代使用。我们先创建一个`LazyProperty`类，用作一个修饰器。当它修饰某个特性时，`LazyProperty`惰性地（首次使用时）加载特性，而不是立即进行。`__init__`方法创建两个变量，用作初始化待修饰特性的方法的别名。`method`变量是一个实际方法的别名，`method_name`变量则是该方法名称的别名。为更好理解如何使用这两个别名，可以将其值输出到标准输出（取消注释下面代码中的两个注释行）。

```
class LazyProperty:
    def __init__(self, method):
        self.method = method
        self.method_name = method.__name__
        # print('function overriden: {}'.format(self.fget))
        # print("function's name: {}".format(self.func_name))
```

LazyProperty类实际上是一个描述符（请参考网页[t.cn/RqrYBND]）。描述符（descriptor）是Python中重写类属性访问方法（__get__()、__set__()和__delete__()）的默认行为要使用的一种推荐机制。LazyProperty类仅重写了__set__()，因为这是其需要重写的唯一访问方法。换句话说，我们无需重写所有访问方法。__get__()方法所访问的特性值，正是下层方法想要赋的值，并使用setattr()来手动赋值。__get__()实际做的事情非常简单，就是使用值来替代方法！这意味着不仅特性是惰性加载的，而且仅可以设置一次。我们马上就能看到这意味着什么。同样，取消注释以下代码的的注释行，以得到一些额外信息。

```
def __get__(self, obj, cls):
    if not obj:
        return None
    value = self.method(obj)
    # print('value {}'.format(value))
    setattr(obj, self.method_name, value)
    return value
```

Test类演示了我们可以如何使用LazyProperty类。其中有三个属性，x、y和_resource。我们想懒加载_resource变量，因此将其初始化为None，如以下代码所示。

```
class Test:
    def __init__(self):
        self.x = 'foo'
        self.y = 'bar'
        self._resource = None
```

resource()方法是使用LazyProperty类修饰的。因演示目的，LazyProperty类将_resource属性初始化为一个tuple，如以下代码所示。通常来说这是一个缓慢/代价大的初始化过程（初始化数据库、图形等）。

```
    @LazyProperty
    def resource(self):
        print('initializing self._resource which is: {}'.format(self._resource))
        self._resource = tuple(range(5))    # 假设这一行的计算成本比较大
        return self._resource
```

main()函数展示了懒初始化是如何进行的。注意，__get__()访问方法的重写使得可以将resource()方法当作一个变量（可以使用t.resource替代t.resource()）。

```
def main():
    t = Test()
    print(t.x)
    print(t.y)
```

```
    # 做更多的事情……
    print(t.resource)
    print(t.resource)
```

从这个例子（文件lazy.py）的执行输出中，可以看出以下几点。

- _resource变量实际不是在t实例创建时初始化的，而是在我们首次使用t.resource时。
- 第二次使用t.resource之时，并没有再次初始化变量。这就是为什么初始化字符串initializing self._resource which is:仅出现一次的原因。
- 下面显示的是lazy.py文件的执行。

```
>>> python3 lazy.py
foo
bar
initializing self._resource which is: None
(0, 1, 2, 3, 4)
(0, 1, 2, 3, 4)
```

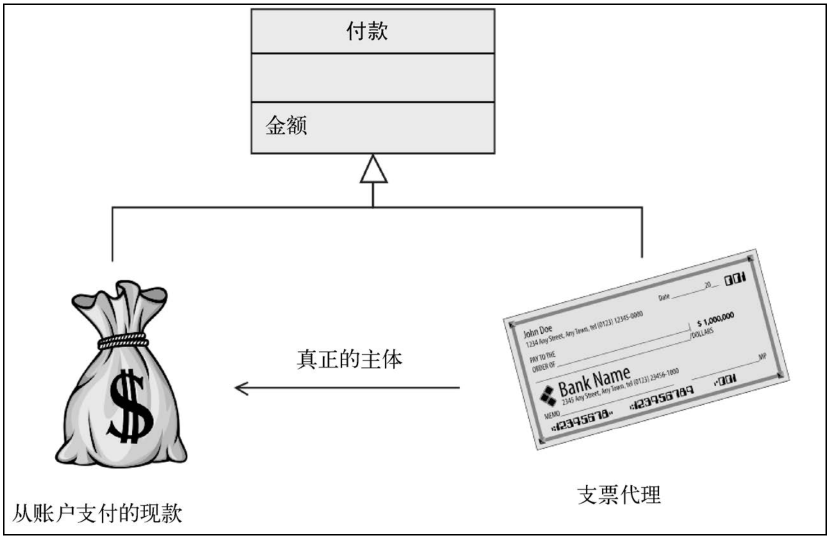
在OOP中有两种基本的、不同类型的懒初始化，如下所示。

- *在实例级*：这意味着会一个对象的特性进行懒初始化，但该特性有一个对象作用域。同一个类的每个实例（对象）都有自己的（不同的）特性副本。
- *在类级或模块级*：在这种情况下，我们不希望每个实例都有一个不同的特性副本，而是所有实例共享同一个特性，而特性是懒初始化的。这一情况在本章不涉及。如果你觉得有意思，可以将其作为练习。

9.1　现实生活的例子

芯片（又名芯片密码）卡（请参考网页［t.cn/RqrYdYx］）是现实生活中使用防护代理的一个好例子。借记/信用卡包含一个芯片，ATM机或读卡器需要先读取芯片；在芯片通过验证后，需要一个密码（PIN）才能完成交易。这意味着只有在物理地提供芯片卡并且知道密码时才能进行交易。

使用银行支票替代现金进行购买和交易是远程代理的一个例子。支票准许了对一个银行账户的访问。下图展示了支票如何用作一个远程代理（请参考网页［t.cn/RqrYEn9］），经sourcemaking.com允许使用。



9.2　软件的例子

Python的`weakref`模块包含一个`proxy()`方法，该方法接受一个输入对象并将一个智能代理返回给该对象。弱引用是为对象添加引用计数支持的一种推荐方式（请参考网页[t.cn/RqrT7cC]）。

ZeroMQ（请参考网页［t.cn/zWzWCrR］）是一组专注于分布式计算的自由开源软件项目。ZeroMQ的Python实现有一个代理模块，实现了一个远程代理。该模块允许Tornado（请参考网页［t.cn/RhFErfr］）的处理程序在不同的远程进程中运行（请参考网页［t.cn/RqrTbY9］）。

9

9.3　应用案例

因为存在至少四种常见的代理类型，所以代理设计模式有很多应用案例，如下所示。

- 在使用私有网络或云搭建一个分布式系统时。在分布式系统中，一些对象存在于本地内存中，一些对象存在于远程计算机的内存中。如果我们不想本地代码关心两者之间的区别，那么可以创建一个远程代理来隐藏/封装，使得应用的分布式性质透明化。
- 因过早创建计算成本较高的对象导致应用遭受性能问题之时。使用虚拟代理引入懒初始化，仅在真正需要对象之时才创建，能够明显提高性能。
- 用于检查一个用户是否有足够权限来访问某个信息片段。如果应用要处理敏感信息（例如，医疗数据），我们会希望确保用户在被准许之后才能访问/修改数据。一个保护/防护代理可以处理所有安全相关的行为。
- 应用（或库、工具集、框架等）使用多线程，而我们希望把线程安全的重任从客户端代

码转移到应用。这种情况下，可以创建一个智能代理，对客户端隐藏线程安全的复杂性。

- 对象关系映射（Object-Relational Mapping，ORM）API也是一个如何使用远程代理的例子。包括Django在内的许多流行Web框架使用一个ORM来提供类OOP的关系型数据库访问。ORM是关系型数据库的代理，数据库可以部署在任意地方，本地或远程服务器都可以。

9.4　实现

为演示代理模式，我们将实现一个简单的保护代理来查看和添加用户。该服务提供以下两个选项。

- 查看用户列表：这一操作不要求特殊权限。
- 添加新用户：这一操作要求客户端提供一个特殊的密码。

SensitiveInfo类包含我们希望保护的信息。users变量是已有用户的列表。read()方法输出用户列表。add()方法将一个新用户添加到列表中。考虑一下下面的代码。

```
class SensitiveInfo:
    def __init__(self):
        self.users = ['nick', 'tom', 'ben', 'mike']

    def read(self):
        print('There are {} users: {}'.format(len(self.users), ' '.join(self.users)))

    def add(self, user):
        self.users.append(user)
        print('Added user {}'.format(user))
```

Info类是SensitiveInfo的一个保护代理。secret变量值是客户端代码在添加新用户时被要求告知/提供的密码。注意，这只是一个例子。现实中，永远不要执行以下操作。

- 在源代码中存储密码
- 以明文形式存储密码
- 使用一种弱（例如，MD5）或自定义加密形式

read()方法是SensetiveInfo.read()的一个包装。add()方法确保仅当客户端代码知道密码时才能添加新用户。考虑一下下面的代码。

```
class Info:
    def __init__(self):
        self.protected = SensitiveInfo()
        self.secret = '0xdeadbeef'

    def read(self):
        self.protected.read()

    def add(self, user):
```

```
        sec = input('what is the secret? ')
        self.protected.add(user) if sec == self.secret else print("That's wrong!")
```

main()函数展示了客户端代码可以如何使用代理模式。客户端代码创建一个Info类的实例，并使用菜单让用户选择来读取列表、添加新用户或退出应用。考虑一下下面的代码。

```
def main():
    info = Info()

    while True:
        print('1. read list |==| 2. add user |==| 3. quit')
        key = input('choose option: ')
        if key == '1':
            info.read()
        elif key == '2':
            name = input('choose username: ')
            info.add(name)
        elif key == '3':
            exit()
        else:
            print('unknown option: {}'.format(key))
```

现在看一下proxy.py文件的完整代码。

```
class SensitiveInfo:
    def __init__(self):
        self.users = ['nick', 'tom', 'ben', 'mike']

    def read(self):
        print('There are {} users: {}'.format(len(self.users), ' '.join(self.users)))

    def add(self, user):
        self.users.append(user)
        print('Added user {}'.format(user))

class Info:
    '''SensitiveInfo的保护代理'''

    def __init__(self):
        self.protected = SensitiveInfo()
        self.secret = '0xdeadbeef'

    def read(self):
        self.protected.read()

    def add(self, user):
        sec = input('what is the secret? ')
        self.protected.add(user) if sec == self.secret else print("That's wrong!")

def main():
    info = Info()

    while True:
        print('1. read list |==| 2. add user |==| 3. quit')
```

9

```
        key = input('choose option: ')
        if key == '1':
            info.read()
        elif key == '2':
            name = input('choose username: ')
            info.add(name)
        elif key == '3':
            exit()
        else:
            print('unknown option: {}'.format(key))
if __name__ == '__main__':
    main()
```

下面是一个如何执行proxy.py的示例。

```
>>> python3 proxy.py
1. read list |==| 2. add user |==| 3. quit
choose option: a
1. read list |==| 2. add user |==| 3. quit
choose option: 4
1. read list |==| 2. add user |==| 3. quit
choose option: 1
There are 4 users: nick tom ben mike
1. read list |==| 2. add user |==| 3. quit
choose option: 2
choose username: pet
what is the secret? blah
That's wrong!
1. read list |==| 2. add user |==| 3. quit
choose option: 2
choose username: bill
what is the secret? 0xdeadbeef
Added user bill
1. read list |==| 2. add user |==| 3. quit
choose option: 1
There are 5 users: nick tom ben mike bill
1. read list |==| 2. add user |==| 3. quit
choose option: 3
```

你已经发现这个代理示例中可以改进的缺陷或缺失特性了吗？我有如下一些建议。

- 该示例有一个非常大的安全缺陷。没有什么能阻止客户端代码通过直接创建一个`SensitveInfo`实例来绕过应用的安全设置。优化示例来阻止这种情况。一种方式是使用`abc`模块来禁止直接实例化`SensitiveInfo`。在这种情况下，会要求进行其他哪些代码变更呢？
- 一个基本的安全原则是，我们绝不应该存储明文密码。只要我们知道使用哪个库，安全地存储密码并不是一件难事（请参考网页［t.cn/zQf0g3c］）。如果你对安全感兴趣，阅读这篇文章，并尝试实现一种外部存储密码的安全方式（例如，在一个文件或数据库中）。
- 应用仅支持添加新用户，那么如何删除一个已有用户呢？添加一个`remove()`方法。`remove()`应该是一个特权操作吗？

9.5 小结

本章中，你学习了如何使用代理设计模式。我们使用代理模式实现一个实际类的替代品，这样可以在访问实际类之前（或之后）做一些额外的事情。存在四种不同的代理类型，如下所示。

- 远程代理，代表一个活跃于远程位置（例如，我们自己的远程服务器或云服务）的对象。
- 虚拟代理，将一个对象的初始化延迟到真正需要使用时进行。
- 保护/防护代理，用于对处理敏感信息的对象进行访问控制。
- 当我们希望通过添加帮助信息（比如，引用计数）来扩展一个对象的行为时，可以使用智能（引用）代理。

在第一个代码示例中，我们使用修饰器和描述符以地道的Python风格创建一个虚拟代理。这个代理允许我们以惰性方式初始化对象属性。

芯片卡和银行支票是人们每天都在使用的两个不同代理的例子。芯片卡是一个防护代理，而银行支票是一个远程代理。另外，一些流行软件中也使用代理。Python有一个`weakref.proxy()`方法，使得创建一个智能代理非常简单。ZeroMQ的Python实现则使用了远程代理。

我们讨论了几个代理模式的应用案例，包括性能、安全及向用户提供简单的API。在第二个代码示例中，我们实现一个保护代理来处理用户信息。这个例子可以以多种方式进行改进，特别是关于其安全缺陷和用户列表实际上未持久化（永久存储）的问题。希望你会发现推荐练习比较有意思。

从第10章开始，我们将探索行为型设计模式。行为型模式处理对象互联和算法的问题。涉及的第一个行为型模式是责任链，我们可以创建一个接收对象的链，从而发送广播消息。在无法预先知道一个请求的处理程序时，发送广播消息非常有用。

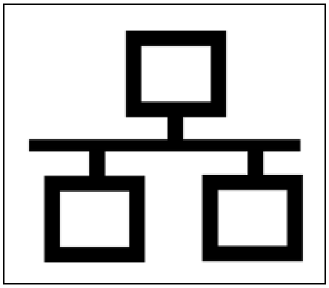
Part 3 第三部分

行为型模式

本部分内容

第 10 章 责任链模式

开发一个应用时，多数时候我们都能预先知道哪个方法能处理某个特定请求。然而，情况并非总是如此。例如，想想任意一种广播计算机网络，例如最早的以太网实现（请参考网页［t.cn/RqrTp0Y］)。在广播计算机网络中，会将所有请求发送给所有节点（简单起见，不考虑广播域），但仅对所发送请求感兴趣的节点会处理请求。加入广播网络的所有计算机使用一种常见的媒介相互连接，比如，下图中的三个节点通过光缆连接起来。



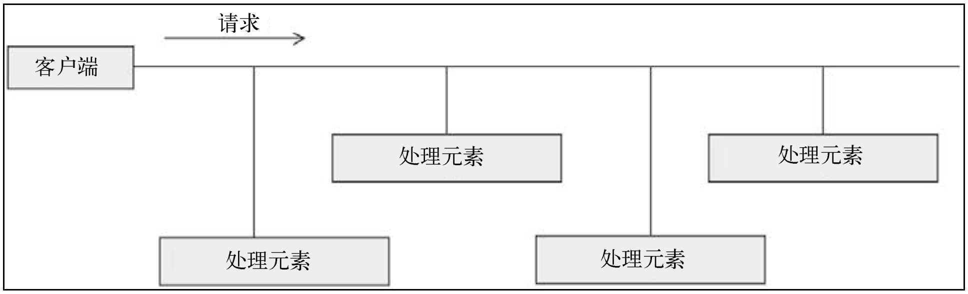
如果一个节点对某个请求不感兴趣或者不知道如何处理这个请求，可以执行以下两个操作。

- 忽略这个请求，什么都不做
- 将请求转发给下一个节点

节点对一个请求的反应方式是实现的细节。然而，我们可以使用广播计算机网络的类比来理解责任链模式是什么。责任链（Chain of Responsibility）模式用于让多个对象来处理单个请求时，或者用于预先不知道应该由哪个对象（来自某个对象链）来处理某个特定请求时。其原则如下所示。

(1) 存在一个对象链（链表、树或任何其他便捷的数据结构）。
(2) 我们一开始将请求发送给链中的第一个对象。
(3) 对象决定其是否要处理该请求。
(4) 对象将请求转发给下一个对象。
(5) 重复该过程，直到到达链尾。

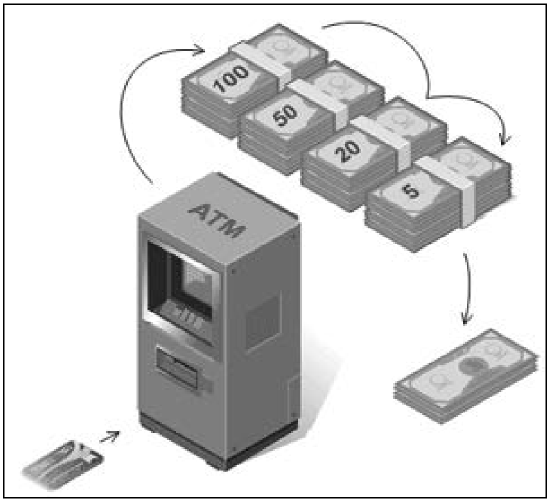
在应用级别，不用讨论光缆和网络节点，而是可以专注于对象以及请求的流程。下图展示了客户端代码如何将请求发送给应用的所有处理元素（又称为节点或处理程序），经www.sourcemaking.com允许使用（请参考网页［t.cn/RqrTYuB］）。



注意，客户端代码仅知道第一个处理元素，而非拥有对所有处理元素的引用；并且每个处理元素仅知道其直接的下一个邻居（称为后继），而不知道所有其他处理元素。这通常是一种单向关系，用编程术语来说是一个单向链表，与之相反的是双向链表。单向链表不允许双向地遍历元素，双向链表则是允许的。这种链式组织方式大有用处：可以解耦发送方（客户端）和接收方（处理元素）（请参考［GOF95，第254页］）。

10.1　现实生活的例子

ATM机以及及一般而言用于接收/返回钞票或硬币的任意类型机器（比如，零食自动贩卖机）都使用了责任链模式。机器上总会有一个放置各种钞票的槽口，如下图所示（经www.sourcemaking.com允许使用）。



钞票放入之后，会被传递到恰当的容器。钞票返回时，则是从恰当的容器中获取（请参考网页［t.cn/RqrTYuB］和网页［t.cn/RqrTnts］）。我们可以把这个槽口视为共享通信媒介，不同的容器则是处理元素。结果包含来自一个或多个容器的现金。例如，在上图中，我们看到在从ATM机取175美元时会发生什么。

10.2　软件的例子

我试过寻找一些使用责任链模式的Python应用的好例子，但是没找到，很可能是因为Python程序员不使用这个名称。因此，很抱歉，我将使用其他编程语言的例子作为参考。

Java的servlet过滤器是在一个HTTP请求到达目标处理程序之前执行的一些代码片段。在使用servlet过滤器时，有一个过滤器链，其中每个过滤器执行一个不同动作（用户身份验证、记日志、数据压缩等），并且将请求转发给下一个过滤器直到链结束；如果发生错误（例如，连续三次身份验证失败）则跳出处理流程（请参考网页［t.cn/RqrTukH］）。

Apple的Cocoa和Cocoa Touch框架使用责任链来处理事件。在某个视图接收到一个其并不知道如何处理的事件时，会将事件转发给其超视图，直到有个视图能够处理这个事件或者视图链结束（请参考网页［t.cn/RqrTrzK］）。

10.3　应用案例

通过使用责任链模式，我们能让许多不同对象来处理一个特定请求。在我们预先不知道应该由哪个对象来处理某个请求时，这是有用的。其中一个例子是采购系统。在采购系统中，有许多核准权限。某个核准权限可能可以核准在一定额度之内的订单，假设为100美元。如果订单超过了100美元，则会将订单发送给链中的下一个核准权限，比如能够核准在200美元以下的订单，等等。

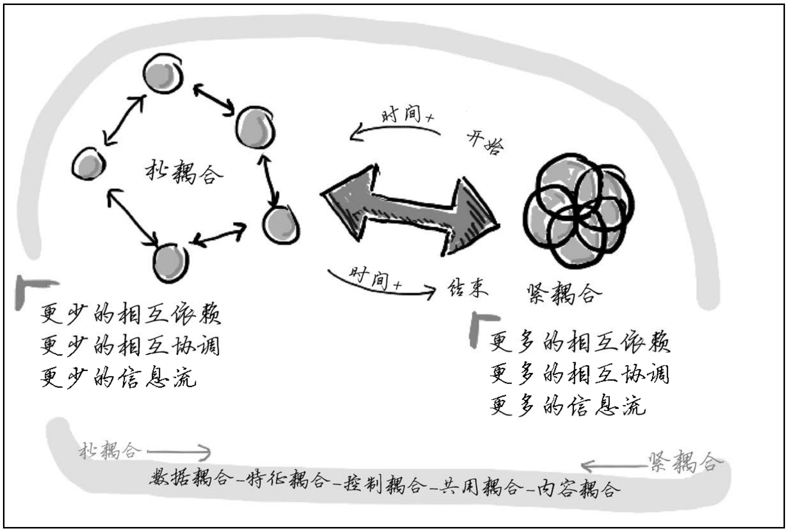
另一个责任链可以派上用场的场景是，在我们知道可能会有多个对象都需要对同一个请求进行处理之时。这在基于事件的编程中是常有的事情。单个事件，比如一次鼠标左击，可被多个事件监听者捕获。

不过应该注意，如果所有请求都能被单个处理程序处理，责任链就没那么有用了，除非确实不知道会是哪个程序处理请求。这一模式的价值在于解耦。客户端与所有处理程序（一个处理程序与所有其他处理程序之间也是如此）之间不再是多对多关系，客户端仅需要知道如何与链的起始节点（标头）进行通信。

下图演示了紧耦合与松耦合之间的区别[①]。松耦合系统背后的考虑是简化维护，并让我们易

① 数据耦合（data coupling）、特征耦合（stamp coupling）、控制耦合（control coupling）、共用耦合（common coupling）和内容耦合（content coupling）这几个概念的含义可参考Wikipedia词条 https://en.wikipedia.org/wiki/Coupling_(computer_programming)。——译者注

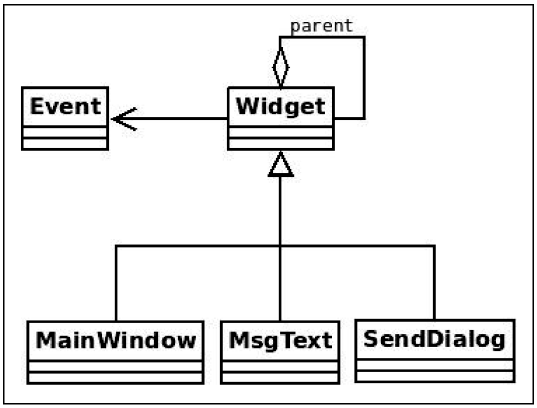
于理解系统的工作原理（请参考网页https://infomgmt.wordpress.com/2010/02/18/a-visual-respresentation-of-coupling/）。



10.4　实现

使用Python实现责任链模式有许多种方式，但是我最喜欢的实现是Vespe Savikko所提出的（请参考网页［t.cn/RqruSj1］）。Vespe的实现以地道的Python风格使用动态分发来处理请求（请参考网页［t.cn/RqruWFp］）。

我们以Vespe的实现为参考实现一个简单的事件系统。下面是该系统的UML类图。



`Event`类描述一个事件。为了让它简单一点，在我们的案例中一个事件只有一个`name`属性。

```
class Event:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name
```

Widget类是应用的核心类。UML图中展示的parent聚合关系表明每个控件都有一个到父对象的引用。按照约定，我们假定父对象是一个Widget实例。然而，注意，根据继承的规则，任何Widget子类的实例（例如，MsgText的实例）也是Widget实例。parent的默认值为None。

```
class Widget:
    def __init__(self, parent=None):
        self.parent = parent
```

handle()方法使用动态分发，通过hasattr()和getattr()决定一个特定请求（event）应该由谁来处理。如果被请求处理事件的控件并不支持该事件，则有两种回退机制。如果控件有parent，则执行parent的handle()方法。如果控件没有parent，但有handle_default()方法，则执行handle_default()。

```
def handle(self, event):
    handler = 'handle_{}'.format(event)
    if hasattr(self, handler):
        method = getattr(self, handler)
        method(event)
    elif self.parent:
        self.parent.handle(event)
    elif hasattr(self, 'handle_default'):
        self.handle_default(event)
```

此时，你可能已明白为什么UML类图中Widget与Event类仅是关联关系而已（不是聚合或组合关系）。关联关系用于表明Widget类知道Event类，但对其没有任何严格的引用，因为事件仅需要作为参数传递给handle()。

MainWindow、MsgText和SendDialog是具有不同行为的控件。我们并不期望这三个控件都能处理相同的事件，即使它们能处理相同事件，表现出来也可能是不同的。MainWindow仅能处理close和default事件。

```
class MainWindow(Widget):
    def handle_close(self, event):
        print('MainWindow: {}'.format(event))

    def handle_default(self, event):
        print('MainWindow Default: {}'.format(event))
```

SendDialog仅能处理paint事件。

```
class SendDialog(Widget):
    def handle_paint(self, event):
```

```
        print('SendDialog: {}'.format(event))
```

最后，MsgText仅能处理down事件。

```
class MsgText(Widget):
    def handle_down(self, event):
        print('MsgText: {}'.format(event))
```

main()函数展示如何创建一些控件和事件，以及控件如何对那些事件作出反应。所有事件都会被发送给所有控件。注意其中每个控件的父子关系。sd对象（SendDialog的一个实例）的父对象是mw（MainWindow的一个实例）。然而，并不是所有对象都需要一个MainWindow实例的父对象。例如，msg对象（MsgText的一个实例）是以sd作为父对象。

```
def main():
    mw = MainWindow()
    sd = SendDialog(mw)
    msg = MsgText(sd)

    for e in ('down', 'paint', 'unhandled', 'close'):
        evt = Event(e)
        print('\nSending event -{}- to MainWindow'.format(evt))
        mw.handle(evt)
        print('Sending event -{}- to SendDialog'.format(evt))
        sd.handle(evt)
        print('Sending event -{}- to MsgText'.format(evt))
        msg.handle(evt)
```

以下是示例的完整代码（chain.py）。

```
class Event:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

class Widget:
    def __init__(self, parent=None):
        self.parent = parent

    def handle(self, event):
        handler = 'handle_{}'.format(event)
        if hasattr(self, handler):
            method = getattr(self, handler)
            method(event)
        elif self.parent:
            self.parent.handle(event)
        elif hasattr(self, 'handle_default'):
            self.handle_default(event)

class MainWindow(Widget):
    def handle_close(self, event):
```

10

```
        print('MainWindow: {}'.format(event))

    def handle_default(self, event):
        print('MainWindow Default: {}'.format(event))

class SendDialog(Widget):
    def handle_paint(self, event):
        print('SendDialog: {}'.format(event))

class MsgText(Widget):
    def handle_down(self, event):
        print('MsgText: {}'.format(event))

def main():
    mw = MainWindow()
    sd = SendDialog(mw)
    msg = MsgText(sd)

    for e in ('down', 'paint', 'unhandled', 'close'):
        evt = Event(e)
        print('\nSending event -{}- to MainWindow'.format(evt))
        mw.handle(evt)
        print('Sending event -{}- to SendDialog'.format(evt))
        sd.handle(evt)
        print('Sending event -{}- to MsgText'.format(evt))
        msg.handle(evt)

if __name__ == '__main__':
    main()
```

执行chain.py会得出以下结果。

```
>>> python3 chain.py

Sending event -down- to MainWindow
MainWindow Default: down
Sending event -down- to SendDialog
MainWindow Default: down
Sending event -down- to MsgText
MsgText: down

Sending event -paint- to MainWindow
MainWindow Default: paint
Sending event -paint- to SendDialog
SendDialog: paint
Sending event -paint- to MsgText
SendDialog: paint

Sending event -unhandled- to MainWindow
MainWindow Default: unhandled
Sending event -unhandled- to SendDialog
MainWindow Default: unhandled
Sending event -unhandled- to MsgText
MainWindow Default: unhandled
```

```
Sending event -close- to MainWindow
MainWindow: close
Sending event -close- to SendDialog
MainWindow: close
Sending event -close- to MsgText
MainWindow: close
```

从输出中我们能看到一些有趣的东西。例如，发送一个`down`事件给`MainWindow`，最终被`MainWindow`默认处理函数处理。另一个不错的用例是，虽然`close`事件不能被`SendDialog`和`MsgText`直接处理，但所有`close`事件最终都能被`MainWindow`正确处理。这正是使用父子关系作为一种回退机制的优美之处。

如果你想在这个事件例子上花费更多时间发挥自己的创意，可以替换这些愚蠢的`print`语句，针对罗列出来的事件添加一些实际的行为。当然，并不限于罗列出来的事件。随意添加一些你喜欢的事件，做一些有用的事情！

另一个练习是在运行时添加一个`MsgText`实例，以`MainWindow`为其父。这个有难度吗？也挑个事件类型来试试（为一个已有控件添加一个新的事件），哪个更难？

10.5 小结

本章中，我们学习了责任链设计模式。在无法预先知道处理程序的数量和类型时，该模式有助于对请求/处理事件进行建模。适合使用责任链模式的系统例子包括基于事件的系统、采购系统和运输系统。

在责任链模式中，发送方可直接访问链中的首个节点。若首个节点不能处理请求，则转发给下一个节点，如此直到请求被某个节点处理或者整个链遍历结束。这种设计用于实现发送方与接收方（多个）之间的解耦。

ATM机是责任链的一个例子。用于取放钞票的槽口可看作是链的头部。从这里开始，根据具体交易，一个或多个容器会被用于处理交易。这些容器可看作是链中的处理程序。

Java的servlet过滤器使用责任链模式对一个HTTP请求执行不同的动作（例如，压缩和身份验证）。Apple的Cocoa框架使用相同的模式来处理事件，比如，按钮和手势。

10.4节演示了在Python中我们可以如何使用动态分发创建基于事件的系统。

第11章介绍命令模式，该模式用于（但不限于）在应用中添加撤销支持。

第 11 章

命令模式

现在多数应用都有撤销操作。虽然难以想象，但在很多年里，任何软件中确实都不存在撤销操作。撤销操作是在1974年引入的（请参考网页［t.cn/Rqr3N22］），但Fortran和Lisp分别早在1957年和1958年就已创建了撤销操作（请参考网页［t.cn/Rqr3067］），这两门语言仍在被人广泛使用。在那些年里，我真心不想使用应用软件。犯了一个错误，用户也没什么便捷方式能修正它。

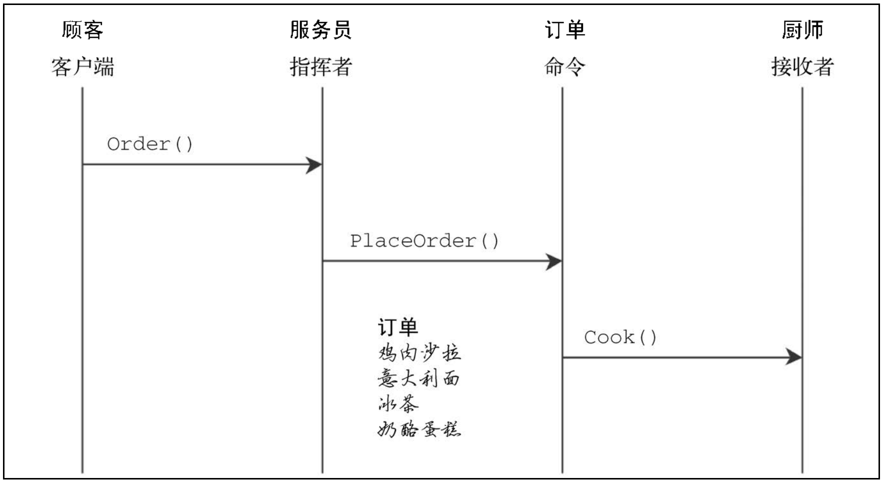
历史就讲到这里。我们想知道如何在应用中实现撤销功能。你已读过本章的标题，所以知道应该推荐哪个设计模式来实现撤销，那就是命令模式（Command pattern）。

命令设计模式帮助我们将一个操作（撤销、重做、复制、粘贴等）封装成一个对象。简而言之，这意味着创建一个类，包含实现该操作所需要的所有逻辑和方法。这样做的优势如下所述（请参考［GOF95，第265页］和网页［t.cn/Rqr3tfQ］）。

- 我们并不需要直接执行一个命令。命令可以按照希望执行。
- 调用命令的对象与知道如何执行命令的对象解耦。调用者无需知道命令的任何实现细节。
- 如果有意义，可以把多个命令组织起来，这样调用者能够按顺序执行它们。例如，在实现一个多层撤销命令时，这是很有用的。

11.1 现实生活的例子

当我们去餐馆吃饭时，会叫服务员来点单。他们用来做记录的账单（通常是纸质的）就是命令模式的一个例子。在记录好订单后，服务员将其放入账单队列，厨师会照着单子去做。每个账单都是独立的，并且可用来执行许多不同命令，例如，一个命令对应一个将要烹饪的菜品。下图展示了一个样例订单的时序图，经www.sourcemaking.com 允许使用（请参考网页［t.cn/Rqr3tfQ］）。



11.2 软件的例子

PyQt是QT工具包的Python绑定。PyQt包含一个`QAction`类，将一个动作建模为一个命令。对每个动作都支持额外的可选信息，比如，描述、工具提示、快捷键和其他（请参考网页［t.cn/Rqr3VQU］）。

git-cola（请参考网页［t.cn/Rqr3IWK］）是使用Python语言编写的一个Git GUI，它使用命令模式来修改模型、变更一次提交、应用一个差异选择、签出，等等（请参考网页［t.cn/Rqr3JVz］）。

11.3 应用案例

许多开发人员以为撤销例子是命令模式的唯一应用案例。撤销操作确实是命令模式的杀手级特性，然而命令模式能做的实际上还有很多（请参考［GOF95，第265页］和网页［t.cn/R4a50r2］）。

- GUI按钮和菜单项：前面提过的PyQt例子使用命令模式来实现按钮和菜单项上的动作。
- 其他操作：除了撤销，命令模式可用于实现任何操作。其中一些例子包括剪切、复制、粘贴、重做和文本大写。
- 事务型行为和日志记录：事务型行为和日志记录对于为变更记录一份持久化日志是很重要的。操作系统用它来从系统崩溃中恢复，关系型数据库用它来实现事务，文件系统用它来实现快照，而安装程序（向导程序）用它来恢复取消的安装。
- 宏：在这里，宏是指一个动作序列，可在任意时间点按要求进行录制和执行。流行的编辑器（比如，Emacs和Vim）都支持宏。

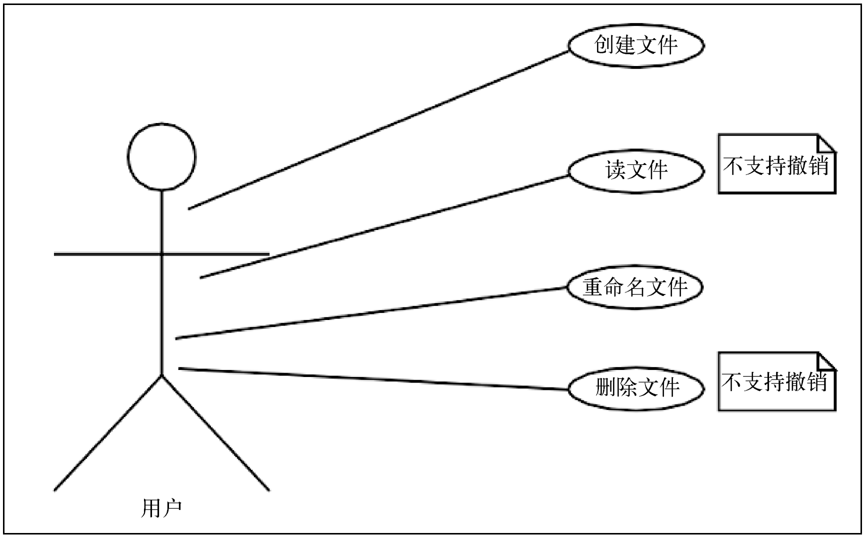
11.4 实现

本节中，我们将使用命令模式实现最基本的文件操作工具。

- 创建一个文件，并随意写入一个字符串
- 读取一个文件的内容
- 重命名一个文件
- 删除一个文件

我们并不从头实现这些工具程序，因为Python在`os`模块中已提供了良好的实现。我们想做的是在已有实现之上添加一个额外的抽象层，这样可以当作命令来使用。这样，我们就能获得命令提供的所有优势。

下面的用例图展示了实现将支持的用户可执行操作。从展示的操作可以看出，重命名文件和创建文件支持撤销。删除一个文件和读取文件内容不支持撤销。对于文件删除操作实际上是可以实现撤销的，一种技术是使用一个特殊的垃圾箱/废物篓目录来存储所有被删除文件，这样在用户请求时可以恢复出来。这是所有现代桌面环境使用的默认行为，就留作练习吧。



每个命令都包括两个部分，初始化部分和执行部分。初始化部分由`__init__()`方法完成，包含该命令发挥作用所要求的所有信息（文件路径和将写入文件的内容等）。执行部分由`execute()`方法完成。在我们想真正地运行命令时才调用其`execute()`方法。该方法并不需要在命令初始化之后立即调用。

我们从重命名工具开始，使用`RenameFile`类来实现。`__init__()`方法接受源文件路径

（path_src）和目标文件路径（path_dest）作为参数。如果文件路径未使用路径分隔符，则在当前目录下创建文件。使用路径分隔符的一个例子是传递字符串/tmp/file1作为path_src，字符串/home/user/file2作为path_dest。不使用路径的例子则是传递file1作为path_src，file2作为path_dest。

```
class RenameFile:
    def __init__(self, path_src, path_dest):
        self.src, self.dest = path_src, path_dest
```

execute()方法使用os.rename()完成实际的重命名。verbose是一个全局标记，被激活时（默认是激活的），能向用户反馈执行的操作。如果你倾向于静默地执行命令，则可以取消激活状态。注意，虽然对于示例来说print()足够好了，但通常会使用更成熟更强大的方式，例如，日志模块（请参考网页［t.cn/Rqr3SXw］）。

```
    def execute(self):
        if verbose:
            print("[renaming '{}' to '{}']".format(self.src, self.dest))
        os.rename(self.src, self.dest)
```

我们的重命名工具通过undo()方法支持撤销操作。在这里，撤销操作再次使用os.rename()将文件名恢复为原始值。

```
    def undo(self):
        if verbose:
            print("[renaming '{}' back to '{}']".format(self.dest, self.src))
        os.rename(self.dest, self.src)
```

文件删除功能实现为单个函数，而不是一个类。我想让你明白并不一定要为想要添加的每个命令（之后会涉及更多）都创建一个新类。delete_file()函数接受一个字符串类型的文件路径，并使用os.remove()来删除它。

```
def delete_file(path):
    if verbose:
        print("deleting file '{}'".format(path))
    os.remove(path)
```

再次回到使用类的方式。CreateFile类用于创建一个文件。__init__()函数接受熟悉的path参数和一个txt字符串，默认向文件写入hello world文本。通常来说，合理的默认行为是创建一个空文件，但因这个例子的需要，我决定向文件写个一个默认字符串。可以根据需要更改它。

```
    def __init__(self, path, txt='hello world\n'):
        self.path, self.txt = path, txt
```

execute()方法使用with语句和open()来打开文件（mode='w'意味着写模式），并使用write()来写入txt字符串。

```
    def execute(self):
```

```
        if verbose:
            print("[Creating file '{}']".format(self.path))
        with open(self.path, mode='w', encoding='utf-8') as out_file:
            out_file.write(self.txt)
```

创建一个文件的撤销操作是删除它。因此，undo()简单地使用delete_file()来实现目的。

```
    def undo(self):
        delete_file(self.path)
```

最后一个工具让我们能够读取文件内容。ReadFile类的execute()方法再次使用with()语句配合open()，这次是读模式，并且只是使用print()来输出文件内容。

```
    def execute(self):
        if verbose:
            print("[reading file '{}']".format(self.path))
        with open(self.path, mode='r', encoding='utf-8') as in_file:
            print(in_file.read(), end='')
```

main()函数使用这些工具类/方法。参数orig_name和new_name是待创建文件的原始名称以及重命名后的新名称。commands列表用于添加（并配置）所有我们之后想要执行的命令。注意，命令不会被执行，除非我们显式地调用每个命令的execute()。

```
    orig_name, new_name = 'file1', 'file2'

    commands = []
    for cmd in CreateFile(orig_name), ReadFile(orig_name), RenameFile(orig_name,
                    new_name):
        commands.append(cmd)

     [c.execute() for c in commands]
```

下一步是询问用户是否需要撤销执行过的命令。用户选择撤销命令或不撤销。如果选择撤销，则执行commands列表中所有命令的undo()。然而，由于并不是所有命令都支持撤销，因此在undo()方法不存在时产生的AttributeError异常要使用异常处理来捕获。如果你不喜欢对这种情况使用异常处理，可以通过添加一个布尔方法（例如，supports_undo()或can_be_undone()）来显式地检测命令是否支持撤销操作。

```
    answer = input('reverse the executed commands? [y/n] ')

    if answer not in 'yY':
        print("the result is {}".format(new_name))
        exit()

    for c in reversed(commands):
        try:
            c.undo()
        except AttributeError as e:
            pass
```

以下是该示例的完整代码（command.py）。

```
import os

verbose = True

class RenameFile:
    def __init__(self, path_src, path_dest):
        self.src, self.dest = path_src, path_dest

    def execute(self):
        if verbose:
            print("[renaming '{}' to '{}']".format(self.src, self.dest))
        os.rename(self.src, self.dest)

    def undo(self):
        if verbose:
            print("[renaming '{}' back to '{}']".format(self.dest, self.src))
        os.rename(self.dest, self.src)

class CreateFile:
    def __init__(self, path, txt='hello world\n'):
        self.path, self.txt = path, txt

    def execute(self):
        if verbose:
            print("[creating file '{}']".format(self.path))
        with open(self.path, mode='w', encoding='utf-8') as out_file:
            out_file.write(self.txt)

    def undo(self):
        delete_file(self.path)

class ReadFile:
    def __init__(self, path):
        self.path = path

    def execute(self):
        if verbose:
            print("[reading file '{}']".format(self.path))
        with open(self.path, mode='r', encoding='utf-8') as in_file:
            print(in_file.read(), end='')

def delete_file(path):
    if verbose:
        print("deleting file '{}'".format(path))
    os.remove(path)

def main():
    orig_name, new_name = 'file1', 'file2'

    commands = []
    for cmd in CreateFile(orig_name), ReadFile(orig_name), RenameFile(orig_name,
```

11

```
new_name):
        commands.append(cmd)

    [c.execute() for c in commands]

    answer = input('reverse the executed commands? [y/n] ')


    if answer not in 'yY':
        print("the result is {}".format(new_name))
        exit()

    for c in reversed(commands):
        try:
            c.undo()
        except AttributeError as e:
            pass

if __name__ == "__main__":
    main()
```

来看看command.py的两次样例执行。第一次不撤销命令，第二次则会撤销。

```
>>> python3 command.py
[creating file 'file1']
[reading file 'file1']
hello world
[renaming 'file1' to 'file2']
reverse the executed commands? [y/n] n
the result is file2

>>> python3 command.py
[creating file 'file1']
[reading file 'file1']
hello world
[renaming 'file1' to 'file2']
reverse the executed commands? [y/n] y
[renaming 'file2' back to 'file1']
deleting file 'file1'
```

这个命令模式的例子可以从多个方面进行改进。首先，这些工具程序都未遵从防御性编程风格（请参考网页［t.cn/Rqr3KHR］）。如果尝试重命名的文件并不存在，那么会发生什么？文件存在但不能对其重命名，因为没有正确的文件系统权限，此时会怎么样？所有工具都存在同样的问题。例如，如果尝试读取一个不存在的文件会发生什么？通过添加一些错误处理逻辑尝试改进这些工具程序。检查os模块方法的返回状态是否必要？

文件创建功能使用默认文件权限来创建文件，默认文件权限具体什么样由文件系统决定。例如，在POSIX系统中，这个权限为-rw-rw-r--。你也许想通过向CreateFile传递恰当的参数让用户能够提供自己的权限设置。可以怎样实现呢？提示，一种方式是通过使用os.fdopen()。

现在，这里有一些东西需要你思考一下。之前我提到过，一个命令并不一定是一个类。文件

删除功能就是那样实现的；仅有一个delete_file()函数。这种方式的优缺点是什么？这里有一个提示，把删除命令放入commands列表，像其余命令那样去执行，可能吗？我们知道在Python中函数是一等公民，因此我们可以执行某些操作，如以下代码所示（文件first-class.py）。

```
orig_name = 'file1'
df = delete_file

commands = []
commands.append(df)

for c in commands:
    try:
        c.execute()
    except AttributeError as e:
        df(orig_name)

for c in reversed(commands):
    try:
        c.undo()
    except AttributeError as e:
        pass
```

虽然这个示例可以工作，但存在以下这些问题。

- 代码不统一。我们过于依赖异常处理，异常处理不是一个程序的常规流程。在这里，所有其他命令都有一个execute()方法，但删除命令没有execute()。
- 目前，文件删除功能还不支持撤销。如果我们最终决定要为其添加撤销支持，那会怎么样呢？通常，我们会为代表命令的那个类添加一个undo()方法。然而，这里的文件删除功能不是类。我们可以创建另一个函数来处理撤销操作，但创建一个类是更好的方式。

11.5　小结

本章中，我们学习了命令模式。使用这种设计模式，可以将一个操作（比如，复制/粘贴）封装为一个对象。这样能提供很多好处，如下所述。

- 我们可以在任何时候执行一个命令，而并不一定是在命令创建时。
- 执行一个命令的客户端代码并不需要知道命令的任何实现细节。
- 可以对命令进行分组，并按一定的顺序执行。

执行一个命令就像在餐馆里点单。每个顾客的订单都是一个独立的命令，分多个阶段，最终由厨师来执行。

许多GUI框架，包括PyQt，使用命令模式来建模动作，动作可被一个或多个事件触发，也可以自定义。然而，命令模式并不仅限于在框架中使用，普通应用（比如git-cola）也会因其而获益。

虽然至今命令模式最广为人知的特性是撤销操作，但它还有更多用处。一般而言，要在运行时按照用户意愿执行的任何操作都适合使用命令模式。命令模式也适用于组合多个命令。这有助于实现宏、多级撤销以及事务。一个事务应该：要么成功，这意味着事务中所有操作应该都成功（提交操作）；要么如果至少一个操作失败，则全部失败（回滚操作）。如果希望进一步使用命令模式，可以实现一个例子，涉及将多个命令组合成一个事务。

为演示命令模式，我们在Python的`os`模块之上实现了一些基本的文件操作工具。我们的工具程序支持撤销，并具有统一的接口，便于组合命令。

第12章将学习解释器模式，该模式可用于创建一种专注于某个特定领域的计算机语言。这种语言被称为领域特定语言（Domain Specific Language，DSL）。

第 12 章 解释器模式

对每个应用来说，至少有以下两种不同的用户分类。

- 基本用户：这类用户只希望能够凭直觉使用应用。他们不喜欢花太多时间配置或学习应用的内部。对他们来说，基本的用法就足够了。
- 高级用户：这些用户，实际上通常是少数，不介意花费额外的时间学习如何使用应用的高级特性。如果知道学会之后能得到以下好处，他们甚至会去学习一种配置（或脚本）语言。
  - 能够更好地控制一个应用
  - 以更好的方式表达想法
  - 提高生产力

解释器（Interpreter）模式仅能引起应用的高级用户的兴趣。这是因为解释器模式背后的主要思想是让非初级用户和领域专家使用一门简单的语言来表达想法。然而，什么是一种简单的语言？对于我们的需求来说，一种简单的语言就是没编程语言那么复杂的语言。

一般而言，我们想要创建的是一种*领域特定语言*（Domain Specific Language，DSL）。DSL是一种针对一个特定领域的有限表达能力的计算机语言。很多不同的事情都使用DSL，比如，战斗模拟、记账、可视化、配置、通信协议等。DSL分为内部DSL和外部DSL（请参考网页［t.cn/zHtEh5t］和网页［t.cn/hBfQ2Y］）。

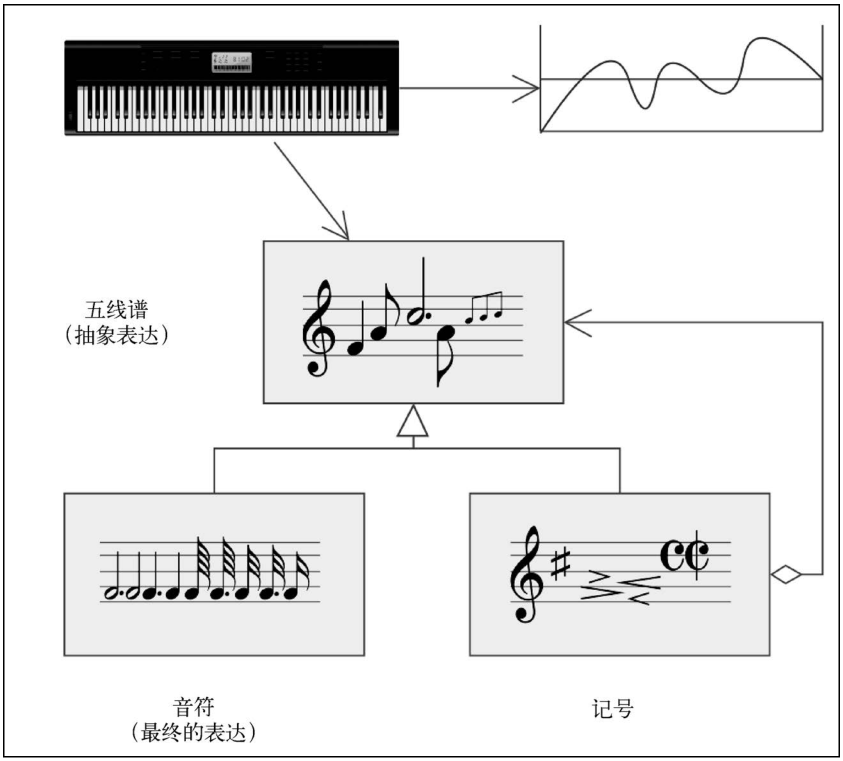
内部DSL构建在一种宿主编程语言之上。内部DSL的一个例子是，使用Python解决线性方程组的一种语言。使用内部DSL的优势是我们不必担心创建、编译及解析语法，因为这些已经被宿主语言解决掉了。劣势是会受限于宿主语言的特性。如果宿主语言不具备这些特性，构建一种表达能力强、简洁而且优美的内部DSL是富有挑战性的（请参考网页［t.cn/Rqr3B12］）。

外部DSL不依赖某种宿主语言。DSL的创建者可以决定语言的方方面面（语法、句法等），但也要负责为其创建一个解析器和编译器。为一种新语言创建解析器和编译器是一个非常复杂、长期而又痛苦的过程（请参考网页［t.cn/Rqr3B12］）。

解释器模式仅与内部DSL相关。因此，我们的目标是使用宿主语言提供的特性构建一种简单但有用的语言，在这里，宿主语言是Python。注意，解释器根本不处理语言解析，它假设我们已经有某种便利形式的解析好的数据，可以是抽象语法树（abstract syntax tree，AST）或任何其他好用的数据结构（请参考［GOF95，第276页］）。

12.1　现实生活的例子

音乐演奏者是现实中解释器模式的一个例子。五线谱图形化地表现了声音的音调和持续时间。音乐演奏者能根据五线谱的符号精确地重现声音。在某种意义上，五线谱是音乐的语言，音乐演奏者是这种语言的解释器。下图展示了音乐例子的图形化描述，经www.sourcemaking.com准许使用（请参考网页［t.cn/Rqr3Fqs］）。



12.2　软件的例子

内部DSL在软件方面的例子有很多。PyT是一个用于生成(X)HTML的Python DSL。PyT关注性能，并声称能与Jinja2的速度相媲美（请参考网页［t.cn/Rqr1vlP］）。当然，我们不能假定在PyT

中必须使用解释器模式。然而，PyT是一种内部DSL，非常适合使用解释器模式。

Chromium是一个自由开源的浏览器软件，催生出了Google Chrome浏览器（请参考网页［t.cn/hMjLK］）。Chromium的Mesa库Python绑定的一部分使用解释器模式将C样板参数翻译成Python对象并执行相关的命令（请参考网页［t.cn/Rqr1zZP］）。

12.3　应用案例

在我们希望为领域专家和高级用户提供一种简单语言来解决他们的问题时，可以使用解释器模式。不过要强调的第一件事情是，解释器模式应仅用于实现简单的语言。如果语言具有外部DSL那样的要求，有更好的工具（yacc和lex、Bison、ANTLR等）来从头创建一种语言。

我们的目标是为专家提供恰当的编程抽象，使其生产力更高；这些专家通常不是程序员。理想情况下，他们使用我们的DSL并不需要了解高级Python知识，当然了解一点Python基础知识会更好，因为我们最终生成的是Python代码，但不应该要求了解Python高级概念。此外，DSL的性能通常不是一个重要的关注点。重点是提供一种语言，隐藏宿主语言的独特性，并提供人类更易读的语法。诚然，Python已经是一门可读性非常高的语言，与其他编程语言相比，其古怪的语法更少。

12.4　实现

我们来创建一种内部DSL控制一个智能屋。这个例子非常契合如今越来越受关注的物联网时代。用户能够使用一种非常简单的事件标记来控制他们的房子。一个事件的形式为`command -> receiver -> arguments`。参数部分是可选的。并不是所有事件都要求参数。不要求任何参数的事件例子如下所示。

```
open -> gate
```

要求参数的事件例子如下所示：

```
increase -> boiler temperature -> 3 degrees
```

`->`符号用于标记事件一个部分的结束，并声明下一个部分的开始。实现一种内部DSL有多种方式。我们可以使用普通的正则表达式、字符串处理、操作符重载的组合以及元编程，或者一个能帮我们完成困难工作的库/工具。虽然在正规情况下，解释器不处理解析，但我觉得一个实战的例子也需要覆盖解析工作。因此，我决定使用一个工具来完成解析工作。该工具名为Pyparsing，是标准Python3发行版的一部分[①]。要想获取更多Pyparsing的信息，可参考Paul McGuire编写的迷

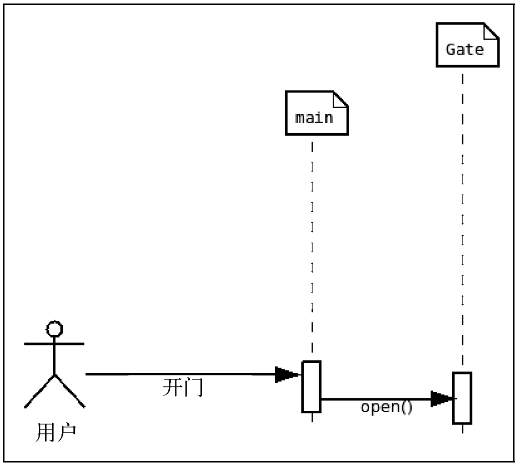
① 这里作者表述有误，Pyparsing并非Python3标准发行版的一部分。——译者注

12

你书*Getting Started with Pyparsing*。如果你的系统上还没安装Pyparsing，可以使用下面的命令来安装。

```
>>> pip3 install pyparsing
```

下面的时序图展示了用户执行开门事件时发生的事情。对于其他事件，情况也是相似的，不过有些命令更复杂些，因为它们要求参数。



在编写代码之前，为我们的语言定义一种简单语法是一个好做法。我们可以使用巴科斯-诺尔形式（Backus-Naur Form，BNF）表示法来定义语法（请参考网页［t.cn/Rqr1ZrO］）。

```
event ::= command token receiver token arguments
command ::= word+
word ::= a collection of one or more alphanumeric characters
token ::= ->
receiver ::= word+
arguments ::= word+
```

简单来说，这个语法告诉我们的是一个事件具有`command -> receiver -> arguments`的形式，并且命令、接收者及参数也具有相同的形式，即一个或多个字母数字字符的组合。包含数字部分是为了让我们能够在命令`increase -> boiler temperature -> 3 degrees`中传递`3 degrees`这样的参数，所以不必怀疑数字部分的必要性。

既然定义了语法，那么接着将其转变成实际的代码。以下是代码的样子。

```
word = Word(alphanums)
command = Group(OneOrMore(word))
token = Suppress("->")
device = Group(OneOrMore(word))
argument = Group(OneOrMore(word))
event = command + token + device + Optional(token + argument)
```

代码和语法定义基本的不同点是，代码需要以自底向上的方式编写。例如，如果不先为word赋一个值，那就不能使用它。Suppress用于声明我们希望解析结果中省略->符号。

这个例子的完整代码（文件interpreter.py）使用了很多占位类，但为了让你精力集中一点，我会先只展示一个类。书中也包含完整的代码列表，在仔细解说完这个类之后会展示。我们来看一下Boiler类。一个锅炉的默认温度为83摄氏度。类有两个方法来分别提高和降低当前的温度。

```
class Boiler:
    def __init__(self):
        self.temperature = 83 # 单位为摄氏度

    def __str__(self):
        return 'boiler temperature: {}'.format(self.temperature)

    def increase_temperature(self, amount):
        print("increasing the boiler's temperature by {} degrees".format(amount))
        self.temperature += amount

    def decrease_temperature(self, amount):
        print("decreasing the boiler's temperature by {} degrees".format(amount))
        self.temperature -= amount
```

下一步是添加语法，之前已学习过。我们也创建一个boiler实例，并输出其默认状态。

```
word = Word(alphanums)
command = Group(OneOrMore(word))
token = Suppress("->")
device = Group(OneOrMore(word))
argument = Group(OneOrMore(word))
event = command + token + device + Optional(token + argument)

boiler = Boiler()
print(boiler)
```

获取Pyparsing解析结果的最简单方式是使用parseString()方法，该方法返回的结果是一个ParseResults实例，它实际上是一个可视为嵌套列表的解析树。例如，执行print(event.parseString('increase -> boiler temperature -> 3 degrees'))得到的结果如下所示。

12

```
[['increase'], ['boiler', 'temperature'], ['3', 'degrees']]
```

因此，在这里，我们知道第一个子列表是命令（提高），第二个子列表是接收者（锅炉温度），第三个子列表是参数（3摄氏度）。实际上我们可以解开ParseResults实例，从而可以直接访问事件的这三个部分。可直接访问意味着我们可以匹配模式找到应该执行哪个方法[①]。

```
    cmd, dev, arg = event.parseString('increase -> boiler temperature -> 3 degrees')
    if 'increase' in ' '.join(cmd):
        if 'boiler' in ' '.join(dev):
```

① 即使不可以直接访问，也能这样做。——译者注

```
            boiler.increase_temperature(int(arg[0]))

    print(boiler)
```

执行上面的代码片段会得到以下输出。

```
>>> python3 boiler.py
boiler temperature: 83
increasing the boiler's temperature by 3 degrees
boiler temperature: 86
```

interpreter.py的完整代码与我刚描述的没有什么大的不同，只是进行了扩展以支持更多的事件和设备。

```
from pyparsing import Word, OneOrMore, Optional, Group, Suppress, alphanums

class Gate:
    def __init__(self):
        self.is_open = False

    def __str__(self):
        return 'open' if self.is_open else 'closed'

    def open(self):
        print('opening the gate')
        self.is_open = True

    def close(self):
        print('closing the gate')
        self.is_open = False

class Garage:
    def __init__(self):
        self.is_open = False

    def __str__(self):
        return 'open' if self.is_open else 'closed'

    def open(self):
        print('opening the garage')
        self.is_open = True

    def close(self):
        print('closing the garage')
        self.is_open = False

class Aircondition:
    def __init__(self):
        self.is_on = False

    def __str__(self):
        return 'on' if self.is_on else 'off'
```

```
    def turn_on(self):
        print('turning on the aircondition')
        self.is_on = True

    def turn_off(self):
        print('turning off the aircondition')
        self.is_on = False

class Heating:
    def __init__(self):
        self.is_on = False

    def __str__(self):
        return 'on' if self.is_on else 'off'

    def turn_on(self):
        print('turning on the heating')
        self.is_on = True

    def turn_off(self):
        print('turning off the heating')
        self.is_on = False

class Boiler:
    def __init__(self):
        self.temperature = 83# in celsius

    def __str__(self):
        return 'boiler temperature: {}'.format(self.temperature)

    def increase_temperature(self, amount):
        print("increasing the boiler's temperature by {} degrees".format(amount))
        self.temperature += amount

    def decrease_temperature(self, amount):
        print("decreasing the boiler's temperature by {} degrees".format(amount))
        self.temperature -= amount

class Fridge:
    def __init__(self):
        self.temperature = 2 # 单位为摄氏度

    def __str__(self):
        return 'fridge temperature: {}'.format(self.temperature)

    def increase_temperature(self, amount):
        print("increasing the fridge's temperature by {} degrees".format(amount))
        self.temperature += amount

    def decrease_temperature(self, amount):
        print("decreasing the fridge's temperature by {} degrees".format(amount))
        self.temperature -= amount

def main():
```

12

```
word = Word(alphanums)
command = Group(OneOrMore(word))
token = Suppress("->")
device = Group(OneOrMore(word))
argument = Group(OneOrMore(word))
event = command + token + device + Optional(token + argument)

gate = Gate()
garage = Garage()
airco = Aircondition()
heating = Heating()
boiler = Boiler()
fridge = Fridge()

tests = ('open -> gate',
         'close -> garage',
         'turn on -> aircondition',
         'turn off -> heating',
         'increase -> boiler temperature -> 5 degrees',
         'decrease -> fridge temperature -> 2 degrees')
open_actions = {'gate':gate.open,
               'garage':garage.open,
               'aircondition':airco.turn_on,
               'heating':heating.turn_on,
               'boiler temperature':boiler.increase_temperature,
               'fridge temperature':fridge.increase_temperature}
close_actions = {'gate':gate.close,
                'garage':garage.close,
                'aircondition':airco.turn_off,
                'heating':heating.turn_off,
                'boiler temperature':boiler.decrease_temperature,
                'fridge temperature':fridge.decrease_temperature}

for t in tests:
    if len(event.parseString(t)) == 2: # 没有参数
        cmd, dev = event.parseString(t)
        cmd_str, dev_str = ' '.join(cmd), ' '.join(dev)
        if 'open' in cmd_str or 'turn on' in cmd_str:
            open_actions[dev_str]()
        elif 'close' in cmd_str or 'turn off' in cmd_str:
            close_actions[dev_str]()
    elif len(event.parseString(t)) == 3: # 有参数
        cmd, dev, arg = event.parseString(t)
        cmd_str, dev_str, arg_str = ' '.join(cmd), ' '.join(dev), ' '.join(arg)
        num_arg = 0
        try:
            num_arg = int(arg_str.split()[0]) # 抽取数值部分
        except ValueError as err:
            print("expected number but got: '{}'".format(arg_str[0]))
        if 'increase' in cmd_str and num_arg > 0:
            open_actions[dev_str](num_arg)
        elif 'decrease' in cmd_str and num_arg > 0:
            close_actions[dev_str](num_arg)
```

```
if __name__ == '__main__':
    main()
```

执行上面的例子会得到以下输出。

```
>>> python3 interpreter.py
opening the gate
closing the garage
turning on the aircondition
turning off the heating
increasing the boiler's temperature by 5 degrees
decreasing the fridge's temperature by 2 degrees
```

如果你想针对这个例子进行更多的实验，我可以给你提一些建议。第一个会让例子更有意思的改变是让其变成交互式。目前，所有事件都是在`tests`元组中硬编码的。然而，用户希望能使用一个交互式提示符来激活命令。不要忘了Pyparsing对空格、Tab或意料之外的输出都是敏感的。例如，如果用户输入`turn     off       -> heating    37`，那会发生什么呢？

另一个可能的改进是，注意`open_actions`和`close_actions`映射是如何用于将一个接收者关联到一个方法的。使用一个映射而不是两个，可能吗？这样做有何优势？

12.5 小结

本章中，我们学习了解释器设计模式。解释器模式用于为高级用户和领域专家提供一个类编程的框架，但没有暴露出编程语言那样的复杂性。这是通过实现一个DSL来达到目的的。

DSL是一种针对特定领域、表达能力有限的计算机语言。DSL有两类，分别是内部DSL和外部DSL。内部DSL构建在一种宿主编程语言之上，依赖宿主编程语言，外部DSL则是从头实现，不依赖某种已有的编程语言。解释器模式仅与内部DSL相关。

乐谱是一个非软件DSL的例子。音乐演奏者像一个解释器那样，使用乐谱演奏出音乐。从软件的视角来看，许多Python模板引擎都使用了内部DSL。PyT是一个高性能的生成(X)HTML的Python DSL。我们也看到Chromium的Mesa库是如何使用解释器模式将图形相关的C代码翻译成Python可执行对象的。

虽然一般来说解释器模式不处理解析的工作，但是在12.4节，我们使用了Pyparsing创建一种DSL来控制一个智能屋，并且看到使用一个好的解析工具以模式匹配来解释结果更加简单。

第13章将演示观察者模式。观察者模式用于在两个或多个对象之间创建一个发布–订阅通信类型。

第 13 章 观察者模式

有时，我们希望在一个对象的状态改变时更新另外一组对象。在MVC模式中有这样一个非常常见的例子，假设在两个视图（例如，一个饼图和一个电子表格）中使用同一个模型的数据，无论何时更改了模型，都需要更新两个视图。这就是观察者设计模式要处理的问题（请参考［Eckel08，第213页］）。

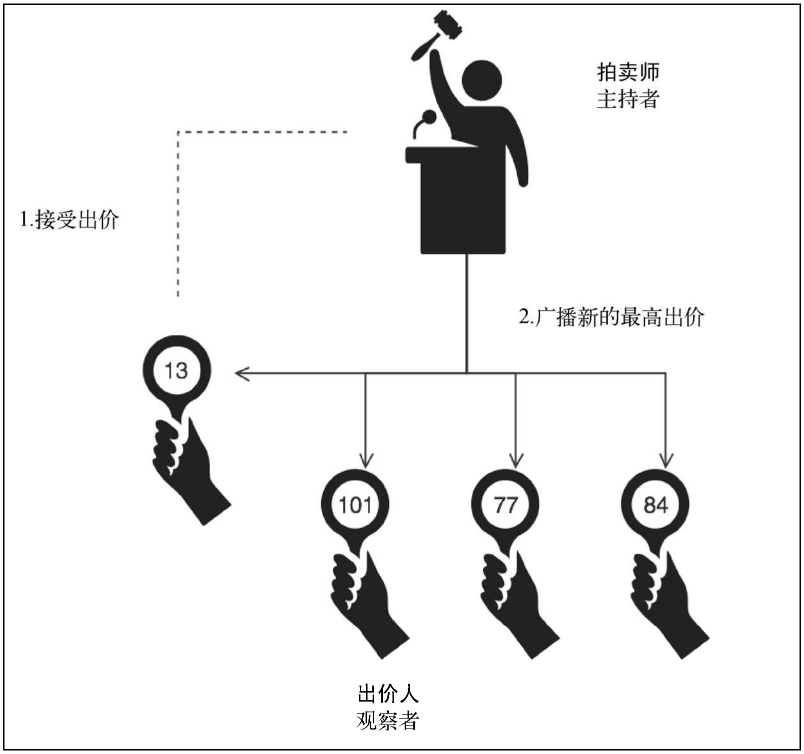
观察者模式描述单个对象（发布者，又称为主持者或可观察者）与一个或多个对象（订阅者，又称为观察者）之间的发布-订阅关系。在MVC例子中，发布者是模型，订阅者是视图。然而，MVC并非是仅有的发布-订阅例子。信息聚合订阅（比如，RSS或Atom）是另一种例子。许多读者通常会使用一个信息聚合阅读器订阅信息流，每当增加一条新信息时，他们就能自动地获取到更新。

观察者模式背后的思想等同于MVC和关注点分离原则背后的思想，即降低发布者与订阅者之间的耦合度，从而易于在运行时添加/删除订阅者。此外，发布者不关心它的订阅者是谁。它只是将通知发送给所有订阅者（请参考［GOF95，第327页］）。

13.1 现实生活的例子

现实中，拍卖会类似于观察者模式。每个拍卖出价人都有一些拍牌，在他们想出价时就可以举起来。不论出价人在何时举起一块拍牌，拍卖师都会像主持者那样更新报价，并将新的价格广播给所有出价人（订阅者）。

下图展示了观察者模式与拍卖会的关联，经www.sourcemaking.com允许使用（请参考网页［t.cn/Rqr1yXo］）。



13.2 软件的例子

django-observer源代码包（请参考网页［t.cn/Rqr14oz］）是一个第三方Django包，可用于注册回调函数，之后在某些Django模型字段发生变化时执行。它支持许多不同类型的模型字段（`CharField`、`IntegerField`等）。

RabbitMQ可用于为应用添加异步消息支持，支持多种消息协议（比如，HTTP和AMQP），可在Python应用中用于实现发布-订阅模式，也就是观察者设计模式（请参考网页［t.cn/Rqr1iIx］）。

13.3 应用案例

当我们希望在一个对象（主持者/发布者/可观察者）发生变化时通知/更新另一个或多个对象的时候，通常会使用观察者模式。观察者的数量以及谁是观察者可能会有所不同，也可以（在运行时）动态地改变。

可以想到许多观察者模式在其中有用武之地的案例。本章开头已提过这样的一个案例，就是信息聚合。无论格式为RSS、Atom还是其他，思想都一样：你追随某个信息源，当它每次更新时，

你都会收到关于更新的一个通知（请参考［Zlobin13，第60页］）。

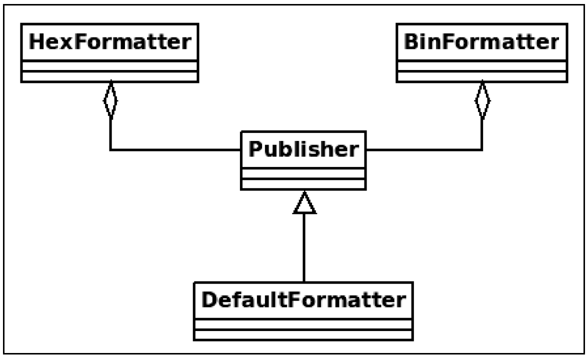
同样的概念也存在于社交网络。如果你使用社交网络服务关联了另一个人，在关联的人更新某些内容时，你能收到相关通知，不论这个关联的人是你关注的一个Twitter用户，Facebook上的一个真实朋友，还是LinkdIn上的一位同事。

事件驱动系统是另一个可以使用（通常也会使用）观察者模式的例子。在这种系统中，监听者被用于监听特定事件。监听者正在监听的事件被创建出来时，就会触发它们。这个事件可以是键入（键盘的）某个特定键、移动鼠标或者其他。事件扮演发布者的角色，监听者则扮演观察者的角色。在这里，关键点是单个事件（发布者）可以关联多个监听者（观察者），请参考网页［t.cn/Rqr1Xgj］。

13.4　实现

本节中，我们将实现一个数据格式化程序。这里描述的想法来源于ActiveState网站上观察者模式用法的Python代码实现（请参考网页［t.cn/Rqr1SDO］）。默认格式化程序是以十进制格式展示一个数值。然而，我们可以添加/注册更多的格式化程序。这个例子中将添加一个十六进制格式化程序和一个二进制格式化程序。每次更新默认格式化程序的值时，已注册的格式化程序就会收到通知，并采取行动。在这里，行动就是以相关的格式展示新的值。

在一些模式中，继承能体现自身价值，观察者模式是这些模式中的一个。我们可以实现一个基类`Publisher`，包括添加、删除及通知观察者这些公用功能。`DefaultFormatter`类继承自`Publisher`，并添加格式化程序特定的功能。我们可以按需动态地添加删除观察者。下面的类图展示了一个使用两个观察者（`HexFormatter`和`BinaryFormatter`）的示例。注意，因为类图是静态的，所以无法展示系统的整个生命周期，只能展示某个特定时间点的系统状态。



从`Publisher`类开始说起。观察者们保存在列表`observers`中。`add()`方法注册一个新的观察者，或者在该观察者已存在时引发一个错误。`remove()`方法注销一个已有观察者，或者在该

观察者尚未存在时引发一个错误。最后，notify()方法则在变化发生时通知所有观察者。

```
class Publisher:
    def __init__(self):
        self.observers = []

    def add(self, observer):
        if observer not in self.observers:
            self.observers.append(observer)
        else:
            print('Failed to add: {}'.format(observer))

    def remove(self, observer):
        try:
            self.observers.remove(observer)
        except ValueError:
            print('Failed to remove: {}'.format(observer))

    def notify(self):
        [o.notify(self) for o in self.observers]
```

接着是DefaultFormatter类。__init__()做的第一件事情就是调用基类的__init__()方法，因为这在Python中没法自动完成。DefaultFormatter实例有自己的名字，这样便于我们跟踪其状态。对于_data变量，我们使用了名称改编来声明不能直接访问该变量。注意，Python中直接访问一个变量始终是可能的（请参考［Lott14，第54页］），不过资深开发人员没有借口这样做，因为代码已经声明不应该这样做。这里使用名称改编是有一个严肃理由的。请继续往下看。DefaultFormatter把_data变量用作一个整数，默认值为零。

```
class DefaultFormatter(Publisher):
    def __init__(self, name):
        Publisher.__init__(self)
        self.name = name
        self._data = 0
```

__str__()方法返回关于发布者名称和_data值的信息。type(self).__name是一种获取类名的方便技巧，避免硬编码类名。这降低了代码的可读性，却提高了可维护性。是否喜欢，要看你的选择。

```
def __str__(self):
        return "{}: '{}' has data = {}".format(type(self).__name__, self.name, self._data)
```

13

类中有两个data()方法。第一个使用@property修饰器来提供_data变量的读访问方式。这样，我们就能使用object.data来替代object.data()。

```
    @property
    def data(self):
        return self._data
```

第二个data()更有意思。它使用了@setter修饰器，该修饰器会在每次使用赋值操作符（=）为_data变量赋新值时被调用。该方法也会尝试把新值强制类型转换为一个整数，并在类型转换

失败时处理异常。

```
    @data.setter
    def data(self, new_value):
        try:
            self._data = int(new_value)
        except ValueError as e:
            print('Error: {}'.format(e))
        else:
            self.notify()
```

下一步是添加观察者。HexFormatter和BinaryFormatter的功能非常相似。唯一的不同在于如何格式化从发布者那获取到的数据值，即分别以十六进制和二进制进行格式化。

```
class HexFormatter:
    def notify(self, publisher):
        print("{}: '{}' has now hex data = {}".format(type(self).__name__,
                    publisher.name, hex(publisher.data)))

class BinaryFormatter:
    def notify(self, publisher):
        print("{}: '{}' has now bin data = {}".format(type(self).__name__,
                    publisher.name, bin(publisher.data)))
```

如果没有测试数据，示例就不好玩了。main()函数一开始创建一个名为test1的Default-Formatter实例，并在之后关联了两个可用的观察者。也使用了异常处理来确保在用户输入问题数据时应用不会崩溃。此外，诸如两次添加相同的观察者或删除尚不存在的观察者之类的事情也不应该导致崩溃。

```
def main():
    df = DefaultFormatter('test1')
    print(df)

    print()
    hf = HexFormatter()
    df.add(hf)
    df.data = 3
    print(df)

    print()
    bf = BinaryFormatter()
    df.add(bf)
    df.data = 21
    print(df)

    print()
    df.remove(hf)
    df.data = 40
    print(df)

    print()
    df.remove(hf)
    df.add(bf)
```

```
    df.data = 'hello'
    print(df)

    print()
    df.data = 15.8
    print(df)
```

示例的完整代码（observer.py）如下所示。

```
class Publisher:
    def __init__(self):
        self.observers = []

    def add(self, observer):
        if observer not in self.observers:
            self.observers.append(observer)
        else:
            print('Failed to add: {}'.format(observer))

    def remove(self, observer):
        try:
            self.observers.remove(observer)
        except ValueError:
            print('Failed to remove: {}'.format(observer))

    def notify(self):
        [o.notify(self) for o in self.observers]

class DefaultFormatter(Publisher):
    def __init__(self, name):
        Publisher.__init__(self)
        self.name = name
        self._data = 0

    def __str__(self):
        return "{}: '{}' has data = {}".format(type(self).__name__, self.name,
self._data)

    @property
    def data(self):
        return self._data

    @data.setter
    def data(self, new_value):
        try:
            self._data = int(new_value)
        except ValueError as e:
            print('Error: {}'.format(e))
        else:
            self.notify()

class HexFormatter:
    def notify(self, publisher):
        print("{}: '{}' has now hex data = {}".format(type(self).__name__,
```

13

```
publisher.name, hex(publisher.data)))

class BinaryFormatter:
    def notify(self, publisher):
        print("{}: '{}' has now bin data = {}".format(type(self).__name__,
publisher.name, bin(publisher.data)))

def main():
    df = DefaultFormatter('test1')
    print(df)

    print()
    hf = HexFormatter()
    df.add(hf)
    df.data = 3
    print(df)

    print()
    bf = BinaryFormatter()
    df.add(bf)
    df.data = 21
    print(df)

    print()
    df.remove(hf)
    df.data = 40
    print(df)

    print()
    df.remove(hf)
    df.add(bf)

    df.data = 'hello'
    print(df)

    print()
    df.data = 15.8
    print(df)

if __name__ == '__main__':
    main()
```

执行observer.py会输出以下内容。

```
>>> python3 observer.py
DefaultFormatter: 'test1' has data = 0

HexFormatter: 'test1' has now hex data = 0x3
DefaultFormatter: 'test1' has data = 3

HexFormatter: 'test1' has now hex data = 0x15
BinaryFormatter: 'test1' has now bin data = 0b10101
DefaultFormatter: 'test1' has data = 21

BinaryFormatter: 'test1' has now bin data = 0b101000
```

```
DefaultFormatter: 'test1' has data = 40

Failed to remove: <__main__.HexFormatter object at 0x7f30a2fb82e8>
Failed to add: <__main__.BinaryFormatter object at 0x7f30a2fb8320>
Error: invalid literal for int() with base 10: 'hello'
BinaryFormatter: 'test1' has now bin data = 0b101000
DefaultFormatter: 'test1' has data = 40

BinaryFormatter: 'test1' has now bin data = 0b1111
DefaultFormatter: 'test1' has data = 15
```

在输出中我们看到，添加额外的观察者，就会出现更多（相关的）输出；一个观察者被删除后，就再也不会被通知到。这正是我们想要的，能够按需启用/禁用运行时通知。

应用的防护性编程方面看起来也工作得不错。尝试玩一些花样都是不会被允许的，比如，删除一个不存在的观察者或者两次添加相同的观察者。不过，显示的信息还不太友好，就留给你作为练习吧。在API要求一个数字参数时输出一个字符串所导致的运行时失败，也能得到正确处理，不会造成应用崩溃/终止。

如果是交互式的，这个例子会有趣得多。即使只是以一个简单的菜单形式允许用户在运行时绑定/解绑观察者或修改`DefaultFormatter`的值，也是不错的，因为这样能看到更多的运行时方面的信息。请随意来做吧。

另一个不错的练习是添加更多的观察者。例如，可以添加一个八进制格式化程序、罗马数字格式化程序或使用你最爱展现形式的任何其他观察者。发挥你的创意，享受乐趣吧！

13.5 小结

本章中，我们学习了观察者设计模式。若希望在一个对象的状态变化时能够通知/提醒所有相关者（一个对象或一组对象），则可以使用观察者模式。观察者模式的一个重要特性是，在运行时，订阅者/观察者的数量以及观察者是谁可能会变化，也可以改变。

为理解观察者模式，你可以想一想拍卖会的场景，出价人是订阅者，拍卖师是发布者。这一模式在软件领域的应用非常多。

大体上，所有利用MVC模式的系统都是基于事件的。作为具体的例子，我们提到了以下两项。

13

- django-observer，一个第三方Django库，用于注册在模型字段变更时执行的观察者。
- RabbitMQ的Python绑定。我们介绍了一个RabbitMQ的具体例子，用于实现发布–订阅（即观察者）模式。

在实现例子中，我们看到了如何使用观察者模式创建可在运行时绑定/解绑的数据格式化程序，以此增强对象的行为。希望你会觉得推荐的练习比较有趣。

第14章介绍状态设计模式，该模式可用于实现一个核心的计算机科学概念：状态机。

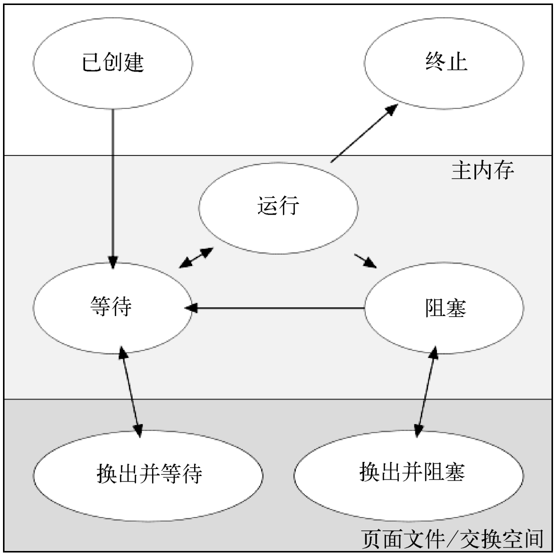
第 14 章

状态模式

面向对象编程着力于在对象交互时改变它们的状态。在很多问题中，有限状态机（通常名为状态机）是一个非常方便的状态转换建模（并在必要时以数学方式形式化）工具。首先，什么是状态机？状态机是一个抽象机器，有两个关键部分，状态和转换。状态是指系统的当前（激活）状况。例如，假设我们有一个收音机，其两个可能的状态是在调频波段（FM）或调幅波段（AM）上调节。另一个可能的状态是从一个FM/AM无线电台切换到另一个。转换是指从一个状态切换到另一个状态，因某个事件或条件的触发而开始。通常，在一次转换发生之前或之后会执行一个或一组动作。假设我们的收音机被调到107 FM无线电台，一次状态转换的例子是收听人按下按钮切换到107.5 FM。

状态机的一个不错的特性是可以用图来表现（称为状态图），其中每个状态都是一个节点，每个转换都是两个节点之间的边。下图展示了一个典型操作系统进程的状态图（不是针对特定的系统），经Wikipedia允许使用（请参考网页［t.cn/Rqr1CDd］）。进程一开始由用户创建好，就进入“已创建/新建”状态。这个状态只能切换到“等待”状态，这个状态转换发生在调度器将进程加载进内存并添加到“等待/预备执行”的进程队列之时。一个“等待”进程有两个可能的状态转换：可被选择而执行（切换到“运行”状态），或被更高优先级的进程所替代（切换到“换出并等待”状态）。

进程的其他典型状态还包括“终止”（已完成或已终止）、“阻塞”（例如，等待一个I/O操作完成）等。需要注意，一个状态机在一个特定时间点只能有一个激活状态。例如，一个进程不可能同时处于“已创建”状态和“运行”状态。



状态机可用于解决多种不同的问题，包括非计算机的问题。非计算机的例子包括自动售货机、电梯、交通灯、暗码锁、停车计时器、自动加油泵及自然语言文法描述。计算机方面的例子包括游戏编程和计算机编程的其他领域、硬件设计、协议设计，以及编程语言解析（请参考网页[t.cn/RUFNdYt]和网页[t.cn/zY5jPeH]）。

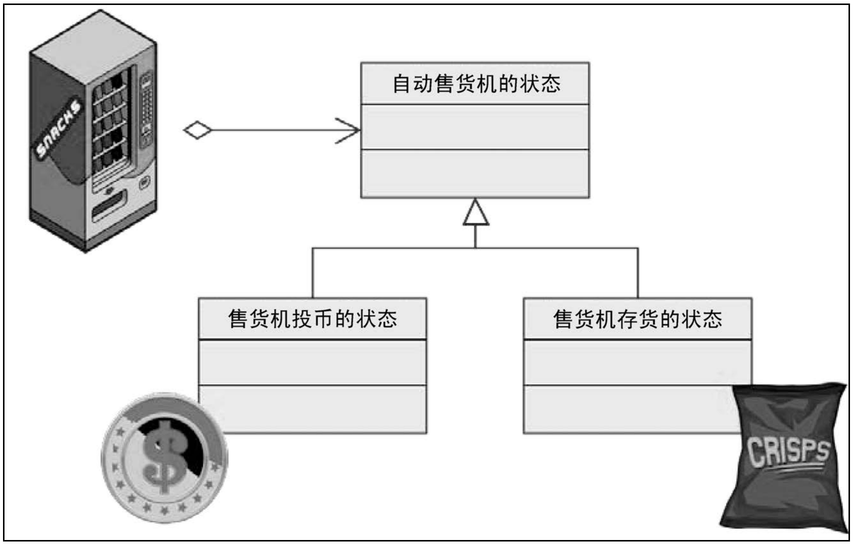
好了，听起来很美好。但是状态机如何关联到状态设计模式（State design pattern）呢？其实状态模式就是应用到一个特定软件工程问题的状态机（请参考[GOF95，第342页]和[Eckel08，第151页]）。

14.1　现实生活的例子

这里再一次提到零食自动售货机（在之前的第10章中见过），它也是日常生活中状态模式的一个例子。自动售货机有不同的状态，并根据我们放入的钱币数量作出不同反应。根据我们的选择和放入的钱币，机器会执行以下操作。

- 拒绝我们的选择，因为请求的货物已售罄。
- 拒绝我们的选择，因为放入的钱币不足。
- 递送货物，且不找零，因为放入的钱币恰好足够。
- 递送货物，并找零。

当然还有更多可能的状态，但你明白要点就好。下图由www.sourcemaking.com提供（请参考网页[t.cn/RqBS3o0]），展示了使用继承实现售货机不同状态的一种可能方案。



14.2　软件的例子

使用状态模式本质上相当于实现一个状态机来解决特定领域的一个软件问题。django-fsm程序包是一个第三方程序包，用于Django框架中简化状态机的实现和使用（请参考网页［t.cn/Rqr1Tgb］）。

Python提供不止一个第三方包/模块来使用和实现状态机（请参考网页［t.cn/Rqr1Qdn］）。我们将在14.4节中看到如何使用其中的一个。

另一个值得一提的项目是状态机编译器（State Machine Compiler，SMC）。使用SMC，你可以使用一种简单的领域特定语言在文本文件中描述你的状态机，SMC会自动生成状态机的代码。该项目声称这种DSL非常简单，写起来就像一对一地翻译一个状态图。我没试过，但听起来非常有意思。SMC可以生成多种编程语言的代码，包括Python（请参考网页［t.cn/RwDrn4v］）。

14.3　应用案例

状态模式适用于许多问题。所有可以使用状态机解决的问题都是不错的状态模式应用案例。我们已经见过的一个例子是操作系统/嵌入式系统的进程模型。

编程语言的编译器实现是另一个好例子。词法和句法分析可使用状态来构建抽象语法树（请参考网页［t.cn/RUFNdYt］）。

事件驱动系统也是另一个例子。在一个事件驱动系统中，从一个状态转换到另一个状态会触

发一个事件/消息。许多计算机游戏都使用这一技术。例如，怪兽会在主人公接近时从防御状态转换到攻击状态（请参考网页［t.cn/Rqr13Lr］和网页［t.cn/Rqr1BW4］)。

这里引用Thomas Jaeger说过的一句话："状态设计模式解决的是一定上下文中无限数量状态的完全封装，从而实现更好的可维护性和灵活性。"（请参考网页［t.cn/8sZrLP0］。)

14.4 实现

下面编写必需的Python代码，演示一下如何基于本章之前提到的状态图创建一个状态机。我们的状态机应该覆盖一个进程的不同状态以及它们之间的转换。

状态设计模式通常使用一个父State类和许多派生的ConcreteState类来实现，父类包含所有状态共同的功能，每个派生类则仅包含特定状态要求的功能。可在网页［t.cn/h47Rs9］上找到一个样例实现。然而在我看来，这些是实现细节。状态模式关注的是实现一个状态机，状态机的核心部分是状态和状态之间的转换。每个部分具体如何实现并不重要。

为避免重复造轮子，可以利用已有的Python模块。它们不仅能帮助我们创建状态机，而且还是地道的Python方式。我发现state_machine这个模块非常有用（请参考网页［t.cn/RqrBvQG］)。在进一步学习之前，如果你的系统上尚未安装state_machine，请使用下面的命令进行安装。

```
>>> pip3 install state_machine
```

state_machine相当简单，不需要特别的介绍。我们将通过示例代码覆盖该模块的大部分内容。

首先从Process类开始。每个创建好的进程都有自己的状态机。使用state_machine模块创建状态机的第一个步骤是使用@acts_as_state_machine修饰器。

```
@acts_as_state_machine
class Process:
```

下一步，定义状态机的状态。这是我们在状态图中看到的节点的映射。唯一的区别是应指定状态机的初始状态。这可通过设置inital=True来指定。

```
    created = State(initial=True)
    waiting = State()
    runnig = State()
    terminated = State()
    blocked = State()
    swapped_out_waiting = State()
    swapped_out_blocked = State()
```

接着定义状态转换。在state_machine模块中，一个状态转换就是一个Event。我们使用参数from_states和to_state来定义一个可能的转换。from_states可以是单个状态或一组状

态（元组）。

```
wait = Event(from_states=(created, running, blocked,
                          swapped_out_waiting), to_state=waiting)
run = Event(from_states=waiting, to_state=running)
terminate = Event(from_states=running, to_state=terminated)
block = Event(from_states=(running, swapped_out_blocked),
                           to_state=blocked)
swap_wait = Event(from_states=waiting, to_state=swapped_out_waiting)
swap_block = Event(from_states=blocked, to_state=swapped_out_blocked)
```

每个进程都有一个名称。正式的应用场景中，一个进程需要多得多的信息才能发挥其作用（例如，ID、优先级和状态等），但为了专注于模式本身，我们进行一些简化。

```
def __init__(self, name):
    self.name = name
```

在发生状态转换时，如果什么影响都没有，那转换就没什么用了。state_machine模块提供@before和@after修饰器，用于在状态转换之前或之后执行动作。为了达到示例的目的，这里的动作限于输出进程状态转换的信息。

```
@after('wait')
def wait_info(self):
    print('{} entered waiting mode'.format(self.name))

@after('run')
def run_info(self):
    print('{} is running'.format(self.name))

@before('terminate')
def terminate_info(self):
    print('{} terminated'.format(self.name))

@after('block')
def block_info(self):
    print('{} is blocked'.format(self.name))

@after('swap_wait')
def swap_wait_info(self):
    print('{} is swapped out and waiting'.format(self.name))

@after('swap_block')
def swap_block_info(self):
    print('{} is swapped out and blocked'.format(self.name))
```

transition()函数接受三个参数：process、event和event_name。process是一个Process类实例，event是一个Event类（wait、run和terminate等）实例，而event_name是事件的名称。在尝试执行event时，如果发生错误，则会输出事件的名称。

```
def transition(process, event, event_name):
    try:
```

```
        event()
    except InvalidStateTransition as err:
        print('Error: transition of {} from {} to {} failed'.format(process.name,
                process.current_state, event_name))
```

state_info()函数展示进程当前（激活）状态的一些基本信息。

```
def state_info(process):
    print('state of {}: {}'.format(process.name, process.current_state))
```

在main()函数的开始，我们定义了一些字符串常量，作为event_name参数值传递。

```
def main():
    RUNNING = 'running'
    WAITING = 'waiting'
    BLOCKED = 'blocked'
    TERMINATED = 'terminated'
```

接着，我们创建两个Process实例，并输出它们的初始状态信息。

```
    p1, p2 = Process('process1'), Process('process2')
    [state_info(p) for p in (p1, p2)]
```

函数的其余部分将尝试不同的状态转换。回忆一下本章之前提到的状态图。允许的状态转换应与状态图一致。例如，从状态“运行”转换到状态“阻塞”是可能的，但从状态“阻塞”转换到状态“运行”则是不可能的。

```
    print()
    transition(p1, p1.wait, WAITING)
    transition(p2, p2.terminate, TERMINATED)
    [state_info(p) for p in (p1, p2)]

    print()
    transition(p1, p1.run, RUNNING)
    transition(p2, p2.wait, WAITING)
    [state_info(p) for p in (p1, p2)]

    print()
    transition(p2, p2.run, RUNNING)
    [state_info(p) for p in (p1, p2)]

    print()
    [transition(p, p.block, BLOCKED) for p in (p1, p2)]
    [state_info(p) for p in (p1, p2)]

    print()
    [transition(p, p.terminate, TERMINATED) for p in (p1, p2)]
    [state_info(p) for p in (p1, p2)]
```

下面是示例的完整代码（文件state.py）。

```
from state_machine import State, Event, acts_as_state_machine, after, before,
InvalidStateTransition
```

```
@acts_as_state_machine
class Process:
    created = State(initial=True)
    waiting = State()
    running = State()
    terminated = State()
    blocked = State()
    swapped_out_waiting = State()
    swapped_out_blocked = State()

    wait = Event(from_states=(created, running, blocked,
                              swapped_out_waiting), to_state=waiting)
    run = Event(from_states=waiting, to_state=running)
    terminate = Event(from_states=running, to_state=terminated)
    block = Event(from_states=(running, swapped_out_blocked),
                               to_state=blocked)
    swap_wait = Event(from_states=waiting, to_state=swapped_out_waiting)
    swap_block = Event(from_states=blocked, to_state=swapped_out_blocked)

    def __init__(self, name):
        self.name = name

    @after('wait')
    def wait_info(self):
        print('{} entered waiting mode'.format(self.name))

    @after('run')
    def run_info(self):
        print('{} is running'.format(self.name))

    @before('terminate')
    def terminate_info(self):
        print('{} terminated'.format(self.name))

    @after('block')
    def block_info(self):
        print('{} is blocked'.format(self.name))

    @after('swap_wait')
    def swap_wait_info(self):
        print('{} is swapped out and waiting'.format(self.name))

    @after('swap_block')
    def swap_block_info(self):
        print('{} is swapped out and blocked'.format(self.name))

def transition(process, event, event_name):
    try:
        event()
    except InvalidStateTransition as err:
        print('Error: transition of {} from {} to {} failed'.format(process.name,
                  process.current_state, event_name))
```

```
def state_info(process):
    print('state of {}: {}'.format(process.name, process.current_state))

def main():
    RUNNING = 'running'
    WAITING = 'waiting'
    BLOCKED = 'blocked'
    TERMINATED = 'terminated'

    p1, p2 = Process('process1'), Process('process2')
    [state_info(p) for p in (p1, p2)]

    print()
    transition(p1, p1.wait, WAITING)
    transition(p2, p2.terminate, TERMINATED)
    [state_info(p) for p in (p1, p2)]

    print()
    transition(p1, p1.run, RUNNING)
    transition(p2, p2.wait, WAITING)
    [state_info(p) for p in (p1, p2)]

    print()
    transition(p2, p2.run, RUNNING)
    [state_info(p) for p in (p1, p2)]

    print()
    [transition(p, p.block, BLOCKED) for p in (p1, p2)]
    [state_info(p) for p in (p1, p2)]

    print()
    [transition(p, p.terminate, TERMINATED) for p in (p1, p2)]
    [state_info(p) for p in (p1, p2)]

if __name__ == '__main__':
    main()
```

下面是执行state.py得到的输出。

```
>>> python3 state.py
state of process1: created
state of process2: created

process1 entered waiting mode
Error: transition of process2 from created to terminated failed
state of process1: waiting
state of process2: created

process1 is running
process2 entered waiting mode
state of process1: running
state of process2: waiting

process2 is running
```

```
state of process1: running
state of process2: running

process1 is blocked
process2 is blocked
state of process1: blocked
state of process2: blocked

Error: transition of process1 from blocked to terminated failed
Error: transition of process2 from blocked to terminated failed
state of process1: blocked
state of process2: blocked
```

确实，输出内容显示，非法的状态转换（比如，“已创建”→“终止”和“阻塞”→“终止”）都失败了。我们不希望应用在请求一个非法转换时崩溃，而except代码块能正确地处理这一点。

注意如何使用stat_machine这样的一个好模块来消除条件式逻辑。没有必要使用冗长易错的if-else语句来检测每个状态转换并作出反应。

为了更好地理解状态模式和状态机，我强烈推荐你实现你自己的例子。可以是任何东西，比如一个简单的电子游戏（你可以使用状态机来处理主人公和敌人的状态）、电梯、解析器或其他任何可以使用状态机来建模的系统。

14.5　小结

本章中，我们学习了状态设计模式。状态模式是一个或多个有限状态机（简称状态机）的实现，用于解决一个特定的软件工程问题。

状态机是一个抽象机器，具有两个主要部分：状态和转换。状态是指一个系统的当前状况。一个状态机在任意时间点只会有一个激活状态。转换是指从当前状态到一个新状态的切换。在一个转换发生之前或之后通常会执行一个或多个动作。状态机可以使用状态图进行视觉上的展现。

状态机用于解决许多计算机问题和非计算机问题，其中包括交通灯、停车计时器、硬件设计和编程语言解析等。我们也看到零食自动贩卖机是如何与状态机的工作方式相关联的。

许多现代软件提供库/模块来简化状态机的实现与使用。Django提供第三方包django-fsm，Python也有许多大家贡献的模块。实际上，在14.4节就使用了其中的一个模块（state_machine）。状态机编译器是另一个有前景的项目，提供许多编程语言的绑定（包括Python）。

我们学习了如何使用state_machine模块为一个计算机系统的进程实现状态机。state_machine模块简化了状态机的创建以及状态转换之前/之后动作的定义。

第15章将学习如何使用策略设计模式实现（在许多候选算法中）动态地选择算法。

第 15 章

策略模式

15

大多数问题都可以使用多种方法来解决。以排序问题为例，对于以一定次序把元素放入一个列表，排序算法有很多。通常来说，没有公认最适合所有场景的算法（请参考网页［t.cn/RqrBZJQ］)。一些不同的评判标准能帮助我们为不同的场景选择不同的排序算法，其中应该考虑的有以下几个。

- 需要排序的元素数量：这被称为输入大小。当输入较少时，几乎所有排序算法的表现都很好，但对于大量输入，只有部分算法具有不错的性能。
- 算法的最佳/平均/最差时间复杂度：时间复杂度是算法运行完成所花费的（大致）时间长短，不考虑系数和低阶项[①]。这是选择算法的最常见标准，但这个标准并不总是那么充分。
- 算法的空间复杂度：空间复杂度是充分地运行一个算法所需要的（大致）物理内存量。在我们处理大数据或在嵌入式系统（通常内存有限）中工作时，这个因素非常重要。
- 算法的稳定性：在执行一个排序算法之后，如果能保持相等值元素原来的先后相对次序，则认为它是稳定的。
- 算法的代码实现复杂度：如果两个算法具有相同的时间/空间复杂度，并且都是稳定的，那么知道哪个算法更易于编码实现和维护也是很重要的。

可能还有更多的评判标准值得考虑，但重要的是，我们真的只能使用单个排序算法来应对所有情况吗？答案当然不是。一个更好的方案是把所有排序算法纳为己用，然后使用上面提到的标准针对当前情况选择最好的算法。这就是策略模式的目的。

策略模式（Strategy pattern）鼓励使用多种算法来解决一个问题，其杀手级特性是能够在运行时透明地切换算法（客户端代码对变化无感知）。因此，如果你有两种算法，并且知道其中一种对少量输入效果更好，另一种对大量输入效果更好，则可以使用策略模式在运行时基于输入数据决定使用哪种算法。

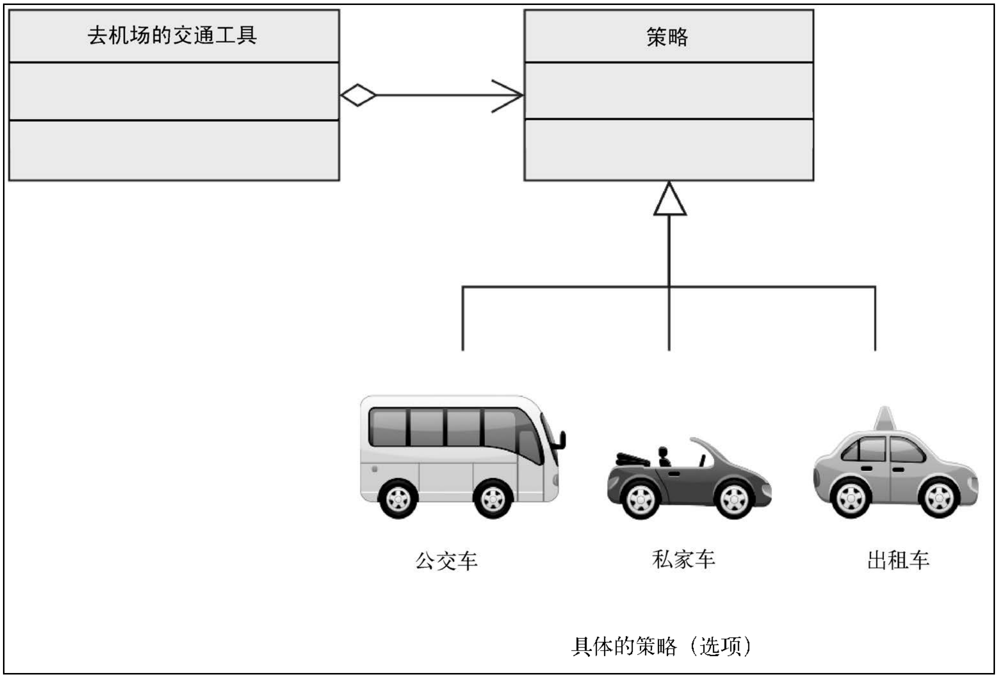
① 在算法分析中，只考虑时间复杂度函数的最高次项，不考虑低阶项，也忽略最高次项的系数。——译者注

15.1　现实生活的例子

去机场赶飞机是现实中使用策略模式的一个恰当例子。

- 如果想省钱，并且早点出发，那么可以坐公交车/地铁。
- 如果不介意支付停车费，并且有自己的汽车，那么可以开车去。
- 如果没有自己的车，又比较急，则可以打车。

这是费用、时间、便利性等因素之间的一个折中权衡。下图展示了以多种方式（策略）去机场的一个例子，经www.sourcemaking.com允许使用（请参考网页［t.cn/RqrBAeJ］）。



15.2　软件的例子

Python的`sorted()`和`list.sort()`函数是策略模式的例子。两个函数都接受一个命名参数`key`，这个参数本质上是实现了一个排序策略的函数的名称（请参考［Eckel08，第202页］）。

下面的例子（代码在文件langs.py中）展示了如何用以下方式使用两种不同的策略对编程语言进行排序。

- 按字母顺序

- 基于它们的流行度（使用TIOBE指数，请参考网页［t.cn/RGQ0jM7］）

namedtuple编程语言（请参考网页［t.cn/RqrBUrf］）用于保存编程语言的统计数据。命名元组是一种易于创建、轻量、不可变的对象类型，与普通元组兼容，但也可以看作一个对象（可以使用常见的类表示法通过名称调用）。命名元组可用于替代以下各项（请参考网页［t.cn/RqrBGwP］）。

- 在我们关注不可变特性时，替代一个类。
- 在值得使用对象表示法来创建可读性更高的代码时，替代一个元组。

顺便说明一下pprint和attrgetter模块。pprint模块用于美化输出一个数据结构，attrgetter用于通过属性名访问class或namedtuple的属性。也可以使用一个lambda函数来替代使用attrgetter，但我觉得attrgetter的可读性更高。

```
import pprint
from collections import namedtuple
from operator import attrgetter

if __name__ == '__main__':
    ProgrammingLang = namedtuple('ProgrammingLang', 'name ranking')

    stats = (('Ruby', 14), ('Javascript', 8), ('Python', 7),
             ('Scala', 31), ('Swift', 18), ('Lisp', 23))

    lang_stats = [ProgrammingLang(n, r) for n, r in stats]
    pp = pprint.PrettyPrinter(indent=5)
    pp.pprint(sorted(lang_stats, key=attrgetter('name')))
    print()
    pp.pprint(sorted(lang_stats, key=attrgetter('ranking')))
```

执行langs.py会得到以下输出。

```
>>>python3 langs.py
[    ProgrammingLang(name='Javascript', ranking=8),
     ProgrammingLang(name='Lisp', ranking=23),
     ProgrammingLang(name='Python', ranking=7),
     ProgrammingLang(name='Ruby', ranking=14),
     ProgrammingLang(name='Scala', ranking=31),
     ProgrammingLang(name='Swift', ranking=18)]

[    ProgrammingLang(name='Python', ranking=7),
     ProgrammingLang(name='Javascript', ranking=8),
     ProgrammingLang(name='Ruby', ranking=14),
     ProgrammingLang(name='Swift', ranking=18),
     ProgrammingLang(name='Lisp', ranking=23),
     ProgrammingLang(name='Scala', ranking=31)]
```


Java API也使用了策略设计模式。java.util.Comparator是一个接口，包含一个compare()方法，该方法本质上是一个策略，可传给排序方法，比如Collections.sort和Arrays.sort

（请参考网页［t.cn/RqrB5o9］）。

15.3　应用案例

策略模式是一种非常通用的设计模式，可应用的场景很多。一般来说，不论何时希望动态、透明地应用不同算法，策略模式都是可行之路。这里所说不同算法的意思是，目的相同但实现方案不同的一类算法。这意味着算法结果应该是完全一致的，但每种实现都有不同的性能和代码复杂性（举例来说，对比一下顺序查找和二分查找）。

我们已看到Python和Java如何使用策略模式来支持不同的排序算法。然而，策略模式并不限于排序问题，也可用于创建各种不同的资源过滤器（身份验证、日志记录、数据压缩和加密等），请参考网页［t.cn/RqrBchI］。

策略模式的另一个应用是创建不同的样式表现，为了实现可移植性（例如，不同平台之间断行的不同）或动态地改变数据的表现。

另一个值得一提的应用是模拟；例如模拟机器人，一些机器人比另一些更有攻击性，一些机器人速度更快，等等。机器人行为中的所有不同之处都可以使用不同的策略来建模（请参考网页［t.cn/RqrBf2q］）。

15.4　实现

关于策略模式的实现没有太多可说的。在函数非一等公民的语言中，每个策略都要用一个不同的类来实现。Wikipedia页面中有UML图展示了这一点（请参考网页［t.cn/RqrBMhW］）。在Python中，我们可以把函数看作是普通的变量，这就简化了策略模式的实现。

假设我们要实现一个算法来检测在一个字符串中是否所有字符都是唯一的。例如，如果输入字符串`dream`，算法应返回true，因为没有字符是重复的。如果输入字符串`pizza`，算法应返回false，因为字母z出现了两次。注意，重复字符不一定是连续的，并且字符串也不一定是一个合法单词。对于字符串`1r2a3ae`，算法也应该返回false，因为其中字母a出现了两次。

在仔细考虑问题之后，我们提出一种实现：对字符串进行排序并逐对比较所有字符。我们首先实现`pairs()`函数，它会返回所有相邻字符对的一个序列`seq`。

```
def pairs(seq):
    n = len(seq)
    for i in range(n):
        yield seq[i], seq[(i + 1) % n]
```

接下来，实现`allUniqueSort()`函数。它接受一个字符串参数`s`，如果该字符串中所有字符都是唯一的，则返回`True`；否则，返回`False`。为演示策略模式，我们进行一些简化，假设这个

算法的伸缩性不好，对于不超过5个字符的字符串才能工作良好。对于更长的字符串，通过插入一条sleep语句来模拟速度减缓。

```
SLOW = 3                # 单位为秒
LIMIT = 5               # 字符数
WARNING = 'too bad, you picked the slow algorithm :('

def allUniqueSort(s):
    if len(s) > LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    srtStr = sorted(s)
    for (c1, c2) in pairs(srtStr):
        if c1 == c2:
            return False
    return True
```

我们对allUniqueSort()的性能并不满意，所以尝试考虑优化的方式。一段时间之后，我们提出一个新算法allUniqueSet()，消除排序的需要。在这里，我们使用一个集合来实现算法。如果正在检测的字符已经被插入到集合中，则意味着字符串中并非所有字符都是唯一的。

```
def allUniqueSet(s):
    if len(s) < LIMIT:
        print(WARNING)
        time.sleep(SLOW)

    return True if len(set(s)) == len(s) else False
```

不幸的是，allUniqueSet()虽然没有伸缩性问题，但出于一些奇怪的原因，它检测短字符串的性能比allUniqueSort()更差。这样的话我们能做点什么呢？没关系，我们可以保留两个算法，并根据待检测字符串的长度来选择最合适的那个算法。函数allUnique()接受一个输入字符串s和一个策略函数strategy，在这里是allUniqueSort()和allUniqueSet()中的一个。函数allUnique执行输入的策略，并向调用者返回结果。

使用main()函数可以执行以下操作。

- 输入待检测字符唯一性的单词
- 选择要使用的策略

该函数还进行了一些基本的错误处理，并让用户能够正常退出程序。

```
def main():
    while True:
        word = None
        while not word:
            word = input('Insert word (type quit to exit)> ')

            if word == 'quit':
                print('bye')
```

15

```
                return

            strategy_picked = None
            strategies = { '1': allUniqueSet, '2': allUniqueSort }
            while strategy_picked not in strategies.keys():
                strategy_picked = input('Choose strategy: [1] Use a set, [2] Sort and
pair> ')

                try:
                    strategy = strategies[strategy_picked]
                    print('allUnique({}): {}'.format(word, allUnique(word,
strategy)))
                except KeyError as err:
                    print('Incorrect option: {}'.format(strategy_picked))
```

下面是该示例的完整代码（文件strategy.py）。

```
import time

SLOW = 3            # 单位为秒
LIMIT = 5           # 字符数
WARNING = 'too bad, you picked the slow algorithm :('

def pairs(seq):
    n = len(seq)
    for i in range(n):
        yield seq[i], seq[(i + 1) % n]

def allUniqueSort(s):
    if len(s) > LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    srtStr = sorted(s)
    for (c1, c2) in pairs(srtStr):
        if c1 == c2:
            return False
    return True

def allUniqueSet(s):
    if len(s) < LIMIT:
        print(WARNING)
        time.sleep(SLOW)

    return True if len(set(s)) == len(s) else False

def allUnique(s, strategy):
    return strategy(s)

def main():
    while True:
        word = None
        while not word:
            word = input('Insert word (type quit to exit)> ')
```

```
            if word == 'quit':
                print('bye')
                return

            strategy_picked = None
            strategies = { '1': allUniqueSet, '2': allUniqueSort }
            while strategy_picked not in strategies.keys():
                strategy_picked = input('Choose strategy: [1] Use a set, [2] Sort and
pair> ')

                try:
                    strategy = strategies[strategy_picked]
                    print('allUnique({}): {}'.format(word, allUnique(word,
strategy)))
                except KeyError as err:
                    print('Incorrect option: {}'.format(strategy_picked))
            print()

if __name__ == '__main__':
    main()
```

我们来看strategy.py的一次样例执行。

```
>>> python3 strategy.py
Insert word (type quit to exit)> balloon
Choose strategy: [1] Use a set, [2] Sort and pair> 1
allUnique(balloon): False

Insert word (type quit to exit)> balloon
Choose strategy: [1] Use a set, [2] Sort and pair> 2
too bad, you picked the slow algorithm :(
allUnique(balloon): False

Insert word (type quit to exit)> bye
Choose strategy: [1] Use a set, [2] Sort and pair> 1
too bad, you picked the slow algorithm :(
allUnique(bye): True

Insert word (type quit to exit)> bye
Choose strategy: [1] Use a set, [2] Sort and pair> 2
allUnique(bye): True


Insert word (type quit to exit)> h
Choose strategy: [1] Use a set, [2] Sort and pair> 1
too bad, you picked the slow algorithm :(
allUnique(h): True

Insert word (type quit to exit)> h
Choose strategy: [1] Use a set, [2] Sort and pair> 2
allUnique(h): False

Insert word (type quit to exit)> quit
bye
```

15

第一个单词（`ballon`）多于5个字符，并且不是所有字符都是唯一的。这种情况下，两个算法都返回了正确结果（`False`），但`allUniqueSort()`更慢，用户也收到了警告。

第二个单词（`bye`）少于5个字符，并且所有字符都是唯一的。再一次，两个算法都返回了期望的结果（`True`），但这一次，`allUniqueSet()`更慢，用户也再一次收到警告。

最后一个单词（`h`）是一个特殊案例。`allUniqueSet()`运行慢，处理正确，返回期望的`True`；算法`allUniqueSort()`返回超快，但结果错误。你能明白为什么吗？作为练习，请修复`allUniqueSort()`算法。你也许想禁止处理单字符的单词，我觉得这样挺不错（相比返回一个错误结果，这样更好）。

通常，我们想要使用的策略不应该由用户来选择。策略模式的要点是可以透明地使用不同的算法。修改一下代码，使得程序始终选择更快的算法。

我们的代码有两种常见用户。一种是最终用户，他们不应该关心代码中发生的事情。为达到这个效果，我们可以遵循前一段给出的提示来实现。另一类用户是其他开发人员。假设我们想创建一个供其他开发人员使用的API。如何做到让他们不用关心策略模式？一个提示是考虑在一个公用类（例如，`AllUnique`）中封装两个函数。这样，其他开发人员只需要创建一个`AllUnique`类实例，并执行单个方法，例如`test()`。在这个方法中需要做些什么呢？

15.5　小结

本章中，我们学习了策略设计模式。策略模式通常用在我们希望对同一个问题透明地使用多种方案时。如果并不存在针对所有输入数据和所有情况的完美算法，那么我们可以使用策略模式，动态地决定在每种情况下应使用哪种算法。现实中，在我们想赶去机场乘飞机时会使用策略模式。

Python使用策略模式让客户端代码决定如何对一个数据结构中的元素进行排序。我们看到了一个例子，基于TIOBE指数排行榜对编程语言进行排序。

策略设计模式的使用并不限于排序领域。加密、压缩、日志记录及其他资源处理的领域都可以使用策略模式来提供不同的数据处理方式。可移植性是策略模式的另一个用武之地。模拟也是另一个策略模式适用的领域。

通过实现两种不同算法来检测一个单词中所有字符的唯一性，我们学习了Python如何因其具有一等函数而简化了策略模式的实现。

在本书的第16章中，我们将学习模板模式。该模式用于抽取一个算法的通用部分，从而提高代码复用。

第 16 章

模板模式

编写优秀代码的一个要素是避免冗余。在面向对象编程中，方法和函数是我们用来避免编写冗余代码的重要工具。回想第15章中的`sorted()`例子。`sorted()`函数非常通用，可使用任意键来对多种数据结构（列表、元组和命名元组等）进行排序。这是一个良好函数的定义。

`sorted()`这样的函数属于理想的案例。现实中，我们没法始终写出100%通用的代码。许多算法都有一些（但并非全部）通用步骤。广度优先搜索（Breadth-First Search，BFS）和深度优先搜索（Depth-First Search，DFS）是其中不错的例子，这两个流行的算法应用于图搜索问题。起初，我们提出独立实现两个算法（文件graph.py）。函数`bfs()`和`dfs()`在`start`和`end`之间存在一条路径时返回一个元组`(True, path)`；如果路径不存在，则返回`(False, path)`（此时，`path`为空）。

```
def bfs(graph, start, end):
        path = []
        visited = [start]
        while visited:
                current = visited.pop(0)
                if current not in path:
                        path.append(current)
                        if current == end:
                                print(path)
                                return (True, path)
                        # 两个顶点不相连，则跳过
                        if current not in graph:
                                continue
                visited = visited + graph[current]
            return (False, path)

def dfs(graph, start, end):
        path = []
        visited = [start]
        while visited:
                current = visited.pop(0)
                if current not in path:
                        path.append(current)
                        if current == end:
                                print(path)
```

```
                    return (True, path)
                # 两个顶点不相连，则跳过
                if current not in graph:
                    continue
            visited = graph[current] + visited
    return (False, path)
```

注意两个算法之间的相似点。仅有一处不同（已加粗），其余部分完全相同。稍后我们再回来讨论这个问题。

先使用Wikipedia提供的图（请参考网页［t.cn/RqrBp3p］）来测试算法。为了简化，假设该图是有向的。这意味着只能朝一个方向移动，我们可以检测如何从Frankfurt到Mannheim，而不是另一个方向。

可以使用列表的字典结构来表示这个有向图。每个城市是字典中的一个键，列表的内容是从该城市始发的所有可能目的地。叶子顶点的城市（例如，Erfurt）使用一个空列表即可（无目的地）。

```
def main():
  graph = {
        'Frankfurt':    ['Mannheim', 'Wurzburg', 'Kassel'],
        'Mannheim': ['Karlsruhe'],
        'Karlsruhe':   ['Augsburg'],
        'Augsburg':  ['Munchen'],
        'Wurzburg':  ['Erfurt', 'Nurnberg'],
        'Nurnberg':  ['Stuttgart', 'Munchen'],
          'Kassel':     ['Munchen'],
        'Erfurt':      [],
        'Stuttgart':  [],
        'Munchen':  []
          }

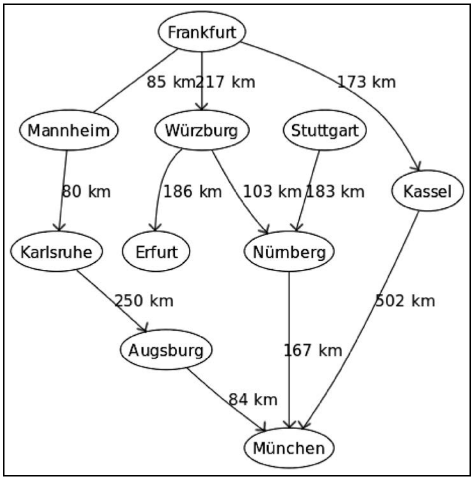
    bfs_path = bfs(graph, 'Frankfurt', 'Nurnberg')
    dfs_path = dfs(graph, 'Frankfurt', 'Nurnberg')
    print('bfs Frankfurt-Nurnberg: {}'.format(bfs_path[1] if bfs_path[0]
  else 'Not
           found'))
    print('dfs Frankfurt-Nurnberg: {}'.format(dfs_path[1] if dfs_path[0]
  else 'Not
          found'))

    bfs_nopath = bfs(graph, 'Wurzburg', 'Kassel')
    print('bfs Wurzburg-Kassel: {}'.format(bfs_nopath[1] if bfs_
  nopath[0] else
          'Not found'))
    dfs_nopath = dfs(graph, 'Wurzburg', 'Kassel')
    print('dfs Wurzburg-Kassel: {}'.format(dfs_nopath[1] if dfs_
  nopath[0] else
           'Not found'))
```

```
if __name__ == '__main__':
  main()
```

从性质来看，结果并不能表明什么，因为DFS和BFS不能很好地处理加权图（权重完全被忽略了）。处理加权图更好的算法是（Dijkstra的）最短路径优先算法、Bellman-Ford算法和A*算法等。然而，我们仍然希望按打算的那样遍历图。我们期望的算法输出是一个城市列表，这些城市是在搜索从Frankfurt到Nurnberg的路径时访问过的。

```
>> python3 graph.py
bfs Frankfurt-Nurnberg: ['Frankfurt', 'Mannheim', 'Wurzburg', 'Kassel', 'Karlsruhe',
                         'Erfurt', 'Nurnberg']
dfs Frankfurt-Nurnberg: ['Frankfurt', 'Mannheim', 'Karlsruhe', 'Augsburg', 'Munchen',
                         'Wurzburg', 'Erfurt', 'Nurnberg']
bfs Wurzburg-Kassel: Not found
dfs Wurzburg-Kassel: Not found
```



结果看起来没问题。BFS按广度进行遍历，DFS则按深度进行遍历，两个算法都没返回任何非期望的结果。这样不错，但我们的代码仍然有一个问题，那就是冗余。两个算法之间仅有一处不同，但其余代码都写了两遍。对于这个问题我们能做点什么吗？

是的！这个问题可以通过模板设计模式（Template design pattern）来解决。这个模式关注的是消除代码冗余，其思想是我们应该无需改变算法结构就能重新定义一个算法的某些部分。为了避免重复而进行必要的重构之后，我们来看看代码会变成什么样子（文件graph_template.py）。

```
def traverse(graph, start, end, action):
        path = []
        visited = [start]
        while visited:
                current = visited.pop(0)
                if current not in path:
```

```
            path.append(current)
            if current == end:
                return (True, path)
            # 两个顶点之间没有连接，则跳过
                if current not in graph:
                    continue
            visited = action(visited, graph[current])
        return (False, path)

def extend_bfs_path(visited, current):
  return visited + current

def extend_dfs_path(visited, current):
  return current + visited
```

不再有bfs()和dfs()两个函数,我们将代码重构为使用单个traverse()函数。traverse()函数实际上是一个模板函数。它接受一个参数action，该参数是一个“知道”如何延伸路径的函数。根据要使用的算法，我们可以传递extend_bfs_path()或extend_dfs_path()作为目标动作。

你也许会争论说，通过在traverse()内部添加一个条件来检测应该使用哪个遍历算法，也能达到相同的结果。下面的代码展示了这个思路（文件graph_template_slower.py）。

```
BFS = 1
DFS = 2

def traverse(graph, start, end, algorithm):
        path = []
        visited = [start]
        while visited:
                current = visited.pop(0)
                if current not in path:
                        path.append(current)
                        if current == end:
                                return (True, path)
                        # 顶点不相连，则跳过
                        if current not in graph:
                                continue
                if algorithm == BFS:
                    visited = extend_bfs_path(visited, graph[current])
                elif algorithm == DFS:
                    visited = extend_dfs_path(visited, graph[current])
                else:
                    raise ValueError("No such algorithm")
        return (False, path)
```

我不喜欢这个方案，有以下几个原因。

- 它使得traverse()难以维护。如果添加第三种方式来延伸路径,就需要扩展traverse()的代码，再添加一个条件来检测是否使用新的路径延伸动作。更好的方案是traverse()

能发挥作用却好像根本不知道应该执行哪个action，因为这样在traverse()中不要求什么特殊逻辑。

- 它仅对只有一行区别的算法有效。如果存在更多区别，那么与让本应归属action的具体细节污染traverse()函数相比，创建一个新函数会好得多。
- 它使得traverse()更慢。这是因为每次traverse()执行时，都需要显式地检测应该执行哪个遍历函数。

执行traverse()与执行dfs()或bfs()没什么大的不同。下面是一个示例。

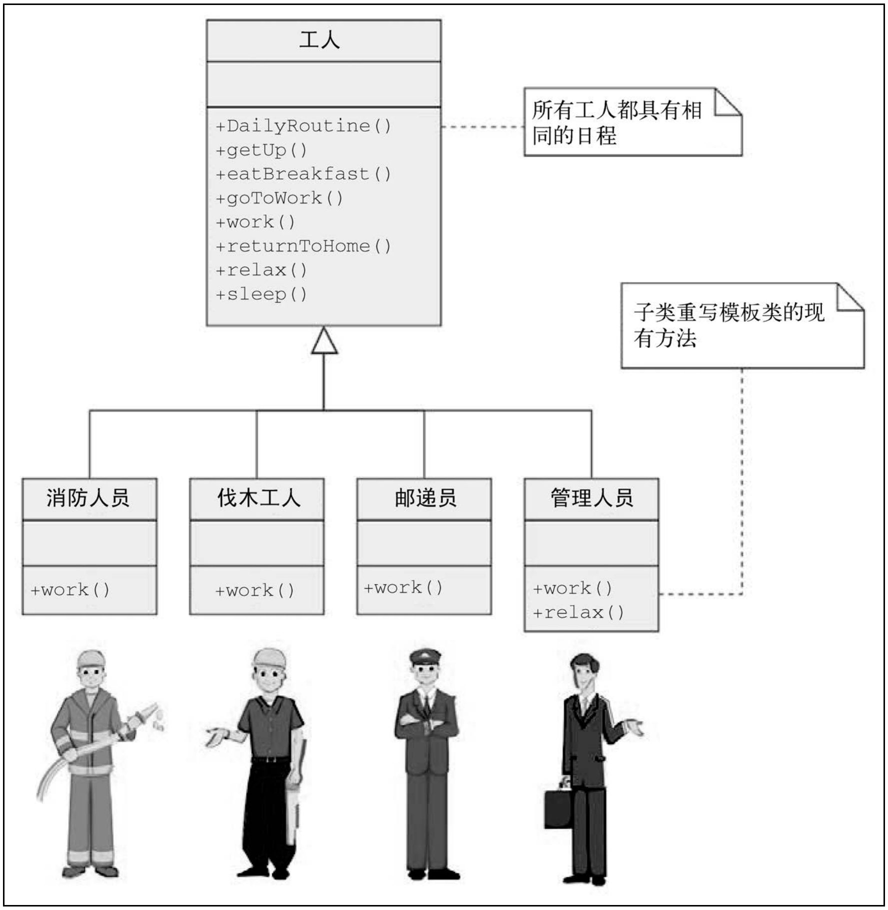
```
bfs_path = traverse(graph, 'Frankfurt', 'Nurnberg', extend_bfs_path)
dfs_path = traverse(graph, 'Frankfurt', 'Nurnberg', extend_dfs_path)
print('bfs Frankfurt-Nurnberg: {}'.format(bfs_path[1] if bfs_path[0] else 'Not
        found'))
print('dfs Frankfurt-Nurnberg: {}'.format(dfs_path[1] if dfs_path[0] else 'Not
        found'))
```

执行graph-template.py的结果应该与执行graph.py的结果相同。

```
>> python3 graph-template.py
bfs Frankfurt-Nurnberg: ['Frankfurt', 'Mannheim', 'Wurzburg', 'Kassel', 'Karlsruhe',
                         'Erfurt', 'Nurnberg']
dfs Frankfurt-Nurnberg: ['Frankfurt', 'Mannheim', 'Karlsruhe', 'Augsburg', 'Munchen',
                         'Wurzburg', 'Erfurt', 'Nurnberg']
bfs Wurzburg-Kassel: Not found
dfs Wurzburg-Kassel: Not found
```

16.1 现实生活的例子

工人的日程，特别是对于同一个公司的工人而言，非常接近于模板设计模式。所有工人都遵从或多或少相同的例行流程，但例行流程的某些特定部分区别又很大。情况如下图所示，该图由www.sourcemaking.com提供（请参考网页［t.cn/RqrBWXo］）。图上展示的模板模式与使用Python实现的模板模式的根本区别在于Python中不强制使用继承。仅在继承对实现有益时，我们才使用它。如果没有实际益处，则可以忽略它，并使用命令和输入惯例。



16.2　软件的例子

Python在cmd模块中使用了模板模式，该模块用于构建面向行的命令解释器。具体而言，cmd.Cmd.cmdloop()实现了一个算法，持续地读取输入命令并将命令分发到动作方法。每次循环之前、之后做的事情以及命令解析部分始终是相同的。这也称为一个算法的不变部分。变化的是实际的动作方法（易变的部分），请参考网页［t.cn/RqrBT6C，第27页］。

Python的asyncore模块也使用了模板模式，该模块用于实现异步套接字服务客户端/服务器。其中诸如asyncore.dispatcher.handle_connect_event和asyncore.dispatcher.handle_write_event()之类的方法仅包含通用代码。要执行特定于套接字的代码，这两个方法会执行handle_connect()方法。注意，执行的是一个特定于套接字的handle_connect()，不是asyncore.dispatcher.handle_connect()。后者仅包含一条警告。可以使用inspect

模块来查看，如下所示。

```
>>> python3
import inspect
import asyncore
inspect.getsource(asyncore.dispatcher.handle_connect)
"    def handle_connect(self):\n        self.log_info('unhandled connect
event', 'warning')\n"
```

16.3 应用案例

模板设计模式旨在消除代码重复。如果我们发现结构相近的（多个）算法中有重复代码，则可以把算法的不变（通用）部分留在一个模板方法/函数中，把易变（不同）的部分移到动作/钩子方法/函数中。

页码标注是一个不错的模板模式应用案例。一个页码标注算法可以分为一个抽象（不变的）部分和一个具体（易变的）部分。不变的部分关注的是最大行号/页号这部分内容。易变的部分则包含用于显示某个已分页特定页面的页眉和页脚的功能（请参考网页［t.cn/RqrBT6C，第10页］)。

所有应用框架都利用了某种形式的模板模式。在使用框架来创建图形化应用时，通常是继承自一个类，并实现自定义行为。然而，在执行自定义行为之前，通常会调用一个模板方法，该方法实现了应用中一定相同的部分，比如绘制屏幕、处理事件循环、调整窗口大小并居中，等等（请参考［EckelPython, 第143页］)。

16.4 实现

本节中，我们将实现一个横幅生成器。想法很简单，将一段文本发送给一个函数，该函数要生成一个包含该文本的横幅。横幅有多种风格，比如点或虚线围绕文本。横幅生成器有一个默认风格，但应该能够使用我们自己提供的风格。

函数generate_banner()是我们的模板函数。它接受一个输入参数（msg，希望横幅包含的文本）和一个可选参数（style，希望使用的风格)。默认风格是dots_style，我们马上就能看到。generate_banner()以一个简单的头部和尾部来包装带样式的文本。实际上，这个头部和尾部可以复杂得多，但在这里调用可以生成头部和尾部的函数来替代仅仅输出简单字符串也无不可。

```
def generate_banner(msg, style=dots_style):
    print('-- start of banner --')
    print(style(msg))
    print('-- end of banner --\n\n')
```

默认的dots_style()简单地将msg首字母大写，并在其之前和之后输出10个点。

```
def dots_style(msg):
    msg = msg.capitalize()
    msg = '.' * 10 + msg + '.' * 10
    return msg
```

该生成器支持的另一个风格是admire_style()。该风格以大写形式展示文本，并在文件的每个字符之间放入一个感叹号。

```
def admire_style(msg):
    msg = msg.upper()
    return '!'.join(msg)
```

接下来这个风格是我目前最喜欢的。cow_style()风格使用cowpy模块生成随机ASCII码艺术字符，夸张地表现文本（请参考网页[t.cn/RqrBnaz]）。如果你的系统中尚未安装cowpy，可以使用下面的命令来安装。

```
>>> pip3 install cowpy
```

cow_style()风格会执行cowpy的milk_random_cow()方法，该方法在每次cow_style()执行时用于生成一个随机的ASCII码艺术字符。

```
def cow_style(msg):
    msg = cow.milk_random_cow(msg)
    return msg
```

main()函数向横幅发送文本“happy coding”，并使用所有可用风格将横幅输出到标准输出。

```
def main():
    msg = 'happy coding'
    [generate_banner(msg, style) for style in (dots_style, admire_style, cow_style)]
```

下面是template.py的完整代码。

```
from cowpy import cow

def dots_style(msg):
    msg = msg.capitalize()
    msg = '.' * 10 + msg + '.' * 10
    return msg

def admire_style(msg):
    msg = msg.upper()
    return '!'.join(msg)

def cow_style(msg):
    msg = cow.milk_random_cow(msg)
    return msg

def generate_banner(msg, style=dots_style):
```

```
    print('-- start of banner --')
    print(style(msg))
    print('-- end of banner --\n\n')

def main():
    msg = 'happy coding'
    [generate_banner(msg, style) for style in (dots_style, admire_style, cow_style)]

if __name__ == '__main__':
    main()
```

下面来看看template.py的一个样例输出。由于cowpy的随机性，你的cow_style()输出也许会有所不同。

```
>>> python3 template.py
-- start of banner --
..........Happy coding..........
-- end of banner --

-- start of banner --
H!A!P!P!Y! !C!O!D!I!N!G
-- end of banner --

-- start of banner --
 ______________
< Happy coding >
 --------------
  \
   \    \_\_ _/_/
    \       \__/
            (xx)\_______
            (__)\       )\/\
             U    ||----w |
                  ||     ||
-- end of banner --
```

你喜欢cowpy生成的艺术字符吗？毫无疑问，我非常喜欢。作为练习，你可以创建自己的风格，并将其应用到横幅生成器。

另一个不错的练习是尝试实现你自己的模板模式例子。找出一些你写过的存在冗余并且模板模式适用的代码。如果从你自己的代码中找不到任何这样的好例子，还可以在GitHub或其他代码托管服务中搜索。找到之后，使用模板模式来重构代码，消除重复。

16.5 小结

本章中，我们学习了模板设计模式。在实现结构相近的算法时，可以使用模板模式来消除冗余代码。具体实现方式是使用动作/钩子方法/函数来完成代码重复的消除，它们是Python中的一等公民。我们学习了一个实际的例子，即使用模板模式来重构BFS和DFS算法的代码。

16

我们看到了一个工人的日常工作是如何与模板模式相类似的，也提到Python标准库中如何使用模板模式的两个例子，还提到了何时使用模板模式的常见应用案例。

本章的最后实现了一个横幅生成器，使用一个模板函数来实现可定制的文本风格。

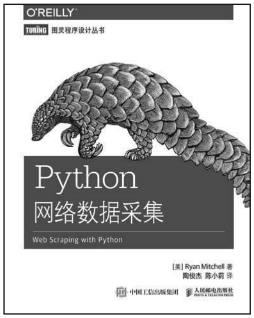
现在到了本书的结尾。我希望阅读本书对你来说是一种享受。最后，借用一位重要的Python贡献者Alex Martelli说过的一句话来提醒你："设计模式是被发现，而不是被发明出来的。"（请参考网页［t.cn/RqrBT6C，第25页］。）

延 展 阅 读



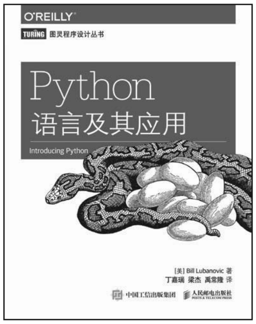
书中内容分为基础篇和实战篇两部分。基础篇介绍基本的编程概念，如列表、字典、类和循环，并指导读者编写整洁且易于理解的代码。另外还介绍了如何让程序能够与用户交互，以及如何在代码运行前进行测试。实战篇介绍如何利用新学到的知识开发功能丰富的项目：2D游戏《外星人入侵》，数据可视化实战，Web应用程序。

书号：978-7-115-42802-8
定价：89.00 元



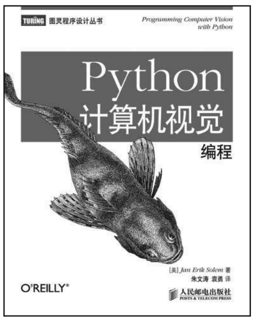
网络上的数据量越来越大，单靠浏览网页获取信息越来越困难，如何有效地提取并利用信息已成为一个巨大的挑战。本书采用简洁强大的Python语言，全面介绍网络数据采集技术，教你从不同形式的网络资源中自由地获取数据。你将学会如何使用Python脚本和网络API一次性采集并处理成千上万个网页上的数据。

书号：978-7-115-41629-2
定价：59.00 元



本书会为你学习Python打下坚实的基础，包括测试、调试、代码复用的最佳实践以及其他开发技巧。同时还会告诉你如何在商业、科学和艺术领域使用Python，并教会你使用多种Python工具和开源包。

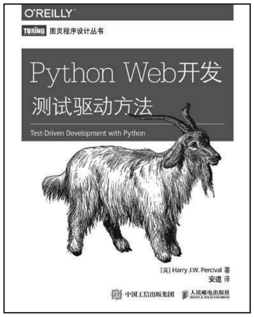
书号：978-7-115-40709-2
定价：79.00 元



本书是计算机视觉编程的权威实践指南，通过Python语言讲解了基础理论与算法，并通过大量示例细致分析了对象识别、基于内容的图像搜索、光学字符识别、光流法、跟踪、3D重建、立体成像、增强现实、姿态估计、全景创建、图像分割、降噪、图像分组等技术。

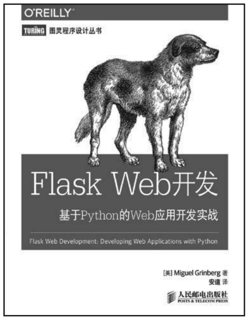
书号：978-7-115-35232-3
定价：69.00 元

延 展 阅 读



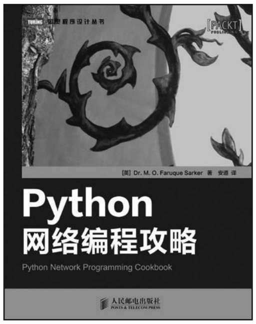
本书手把手教你从头开始开发一个真正的Web应用，并且展示使用Python做测试驱动开发（TDD）的优势。你将学到如何在开发应用的每一个部分之前先编写和运行测试，然后再编写最少量的代码让测试通过。也就是说，你将学会应用TDD理念，写出简洁可用、赏心悦目的代码。

书号：978-7-115-40327-8
定价：99.00 元



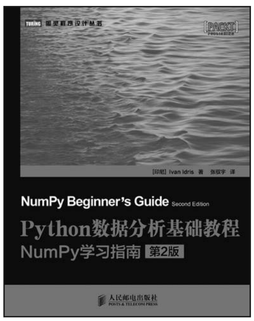
本书共分三部分，全面介绍如何基于Python微框架Flask进行Web开发。第一部分是Flask简介，介绍使用Flask框架及扩展开发Web程序的必备基础知识。第二部分则给出一个实例，真正带领大家一步步开发完整的博客和社交应用Flasky，从而将前述知识融会贯通，付诸实践。第三部分介绍了发布应用之前必须考虑的事项，如单元测试策略、性能分析技术、Flask程序的部署方式等。

书号：978-7-115-37399-1
定价：59.00 元



Python是门很棒的编程语言，适合快速构建应用原型。本书全面介绍了Python网络编程涉及的重要问题，包括网络编程、系统和网络管理、网络监控以及Web应用开发。作者通过70多篇攻略，清晰简明地描述了各种网络任务和问题，提出了可用于多种场景的解决方案，并细致地分析了整个操作过程。无需多少Python基础知识，就可以轻松理解这些示例。如果你想开发依赖于网络协议的实用Web应用和网络应用，绝对不能错过这本书。

书号：978-7-115-37269-7
定价：45.00 元



本书从NumPy安装讲起，逐渐过渡到数组对象、常用函数、矩阵运算、线性代数、金融函数、窗函数、质量控制等内容，致力于向初中级Python编程人员全面讲述NumPy及其使用。另外，通过书中丰富的示例，你还将学会Matplotlib绘图，并结合使用其他Python科学计算库（如SciPy和Scikits），让工作更有成效，让代码更加简洁而高效。

书号：978-7-115-33940-9
定价：49.00 元

本书用实际生活中的例子带你了解常用的设计模式，介绍了诸多有关编写Python风格代码的底层细节和概念，包括故障排除、最佳实践、系统架构和设计原则等，帮你解决每天都会遇到的问题。

- 探索用工厂方法和抽象工厂来创建对象
- 使用原型模式克隆对象
- 使用适配器模式让不兼容的接口变得兼容
- 使用代理模式保障接口安全
- 使用策略模式动态选择算法
- 使用修饰器模式，在不使用子类化的情况下扩展对象
- 使用模型–视图–控制器模式将逻辑从用户界面解耦

Amazon读者推荐

“这本书结构合理、语言简洁，各章自成一体，可以单独阅读。每章都包含一些理论解释以及大量代码和现实生活中的例子。用例对于理解在何时运用何种设计模式很有帮助。如果你是想要学习设计模式的Python程序员，我真诚地向你推荐这本书。”

“如果想要优化代码、增强其可读性，你应该看看这本使用了代码结构示例的书，看后一定会有新的想法。这本书的写作风格干净利落、紧抓要点，不适合初学者，但如果你具备一些Python编程经验，想要学习新的编程风格，这本书很适合你。”

[PACKT] PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/程序设计/Python
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-42803-5
9 787115 428035
ISBN 978-7-115-42803-5
定价：45.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks